
UNIVERSITY OF VERONA

DOCTORAL THESIS

# Formal Approaches to Control System Security

## From Static Analysis to Runtime Enforcement

*Author:*
Andrei MUNTEANU

*Supervisor:*
Prof. Massimo MERRO
*Co-Supervisor:*
Dr. Ruggero LANOTTE

*A thesis submitted in fulfillment of the requirements*
*for the degree of Doctor of Philosophy in*

Computer Science

October 6, 2021

# Declaration

I, Andrei MUNTEANU, declare that this thesis titled, "Formal Approaches to Control System Security: From Static Analysis to Runtime Enforcement", which is submitted in fulfilment of the requirements for the Degree of Doctor of Philosophy, represents my own work except where due acknowledgement have been made. I further declared that it has not been previously included in a thesis, dissertation, or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Andrei Munteanu

Date: October 6, 2021

# *Acknowledgements*

I would like to thank my parents Anton and Aurica for their support and their encouragement in pursuing my academic goals. They are and always will be role models for me and I love them more than words can describe.I would like to thank my sister Maria for always cooking great meals for me. I would like to thank my girlfriend Sara for being a great companion and of course for all her many practical advice. I would like to thank my friends and colleagues. I'm glad that I have shared with them tough moments, great intellectual conversations and many many beers. I thank Professor Paolo Fiorini for his support in developing this work and for introducing me to the world of academic research.

I thank my supervisor Massimo Merro for constantly pushing and encouraging me to go beyond my limits. I would also like to thank him for all those stories he shared with me. I am grateful to have worked with someone with such a great work ethic. Finally, I would like to thank Ruggero Lanotte for his support and for being patient when I didn't understand things as quickly as he did. I'm glad I had the opportunity to work with such a brilliant mind.

Andrei MUNTEANU
University of Verona
October 6, 2021

# Abstract

With the advent of *Industry 4.0*, industrial facilities and critical infrastructures are transforming into an ecosystem of heterogeneous physical and cyber components, such as *programmable logic controllers*, increasingly interconnected and therefore exposed to *cyber-physical attacks*, *i.e.*, security breaches in cyberspace that may adversely affect the physical processes underlying *industrial control systems*.

The main contributions of this thesis follow two research strands that address the security concerns of industrial control systems via *formal methodologies*. As our first contribution, we propose a formal approach based on *model checking* and *statistical model checking*, within the MODEST TOOLSET, to analyse the impact of attacks targeting non-trivial control systems equipped with an intrusion detection system (IDS) capable of *detecting* and *mitigating* attacks. Our goal is to evaluate the impact of *cyber-physical attacks*, *i.e.*, attacks targeting *sensors* and/or *actuators* of the system with potential consequences on the safety of the inner physical process. Our security analysis estimates both the *physical impact* of the attacks and the performance of the IDS.

As our second contribution, we propose a formal approach based on *runtime enforcement* to ensure specification compliance in networks of controllers, possibly compromised by *colluding malware* that may tamper with actuator commands, sensor readings, and inter-controller communications. Our approach relies on an ad-hoc sub-class of Ligatti et al.'s *edit automata* to enforce controllers represented in Hennessy and Regan's *Timed Process Language*. We define a synthesis algorithm that, given an alphabet $\mathcal{P}$ of observable actions and a timed correctness property $e$, returns a monitor that enforces the property $e$ during the execution of any (potentially corrupted) controller with alphabet $\mathcal{P}$, and complying with the property $e$. Our monitors *correct* and *suppress* incorrect actions coming from corrupted controllers and *emit* actions in full autonomy when the controller under scrutiny is not able to do so in a correct manner. Besides classical requirements, such as *transparency* and *soundness*, the proposed enforcement enjoys *deadlock- and diverge-freedom* of monitored controllers, together with *compositionality* when dealing with networks of controllers. Finally, we test the proposed enforcement mechanism on a non-trivial case study, taken from the context of industrial water treatment systems, in which the controllers are injected with different malware with different malicious goals.

# List of Publications

## Journals:

[1] R. Lanotte, M. Merro, A. Munteanu, and L. Viganò. A Formal Approach to Physics-based Attacks in Cyber-physical Systems. ACM Transactions on Privacy and Security (TOPS) 23, 1 (2020), 3:1–3:41.

[2] R. Lanotte, M. Merro, and A. Munteanu. Runtime Enforcement of Programmable Logic Controllers. Submitted to journal.

[3] R. Lanotte, M. Merro, and A. Munteanu. A process calculus approach to detection and mitigation of PLC malware. To appear in the journal Theoretical Computer Science.

## Conferences:

[1] R. Lanotte, M. Merro, and A. Munteanu. 2018. A Modest Security Analysis of Cyber-Physical Systems: A Case Study. In Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Vol. 10854. Springer, 58–78.

[2] A. Munteanu, M. Pasqua, and M. Merro. 2020. Impact Analysis of Cyber-Physical Attacks on a Water Tank System via Statistical Model Checking. In IEEE/ACM 8th Internation Conference on Formal Methods in Sw Eng. (FormaliSE), pp. 34-43.

[3] R. Lanotte, M. Merro, and A. Munteanu. 2020. Runtime Enforcement for Control System Security. In IEEE 33rd Computer Security Foundations Symposium (CSF), 246–261.

[4] R. Lanotte, M. Merro, and A. Munteanu. 2020. A process calculus approach to correctness enforcement of PLCs. In ICTCS (CEUR Workshop Proceedings, Vol. 2756). CEUR-WS.org, 81–94.

[5] R. Lanotte, M. Merro, A. Munteanu, and S. Tini. 2021. Formal Impact Metrics for Cyber-Physical Attacks. To appear in 34th Computer Security Foundations Symposium (CSF).

# Contents

**Declaration** i

**Acknowledgements** ii

**List of Publications** v

**List of Figures** xi

**List of Tables** xiii

**1 Introduction** 1
- 1.1 Our objectives . . . 2
- 1.2 Contribution . . . 5
  - 1.2.1 An impact analysis for the security of control systems . . . 5
  - 1.2.2 Runtime enforcement for control systems security . . . 7
- 1.3 Outline . . . 8

**2 Industrial control systems and their vulnerabilities** 9
- 2.1 An introduction to industrial control systems . . . 9
  - 2.1.1 A closer look to programmable logic controller . . . 12
- 2.2 High profile attacks targeting ICSs . . . 14
- 2.3 Anatomy of cyber-physical attacks targeting ICSs . . . 16
- 2.4 Security measures for ICSs . . . 20
  - 2.4.1 State of the art . . . 22

**3 Technical background: formal verification methodologies** 27
- 3.1 Model checking of hybrid systems . . . 27
  - 3.1.1 Hybrid automata . . . 27
  - 3.1.2 Temporal logics . . . 32
  - 3.1.3 Tools . . . 33
- 3.2 Statistical model checking . . . 33
- 3.3 Runtime enforcement . . . 34
  - 3.3.1 Runtime enforcement: a formal approach . . . 34
  - 3.3.2 Ligatti et al.'s edit automata . . . 35

**PART I: An impact analysis for the security of control systems** 37

**4 A Model Checking Approach** **39**
4.1 A simple engine with a cooling system . . . 39
4.1.1 Analyses and results . . . 45
4.2 Summary and discussion . . . 49

**5 A Statistical Model Checking Approach** **51**
5.1 A quadruple water tank system . . . 51
5.1.1 Analyses and results . . . 58
5.2 Summary and discussion . . . 64

**6 End of Part I** **65**
6.1 Related work . . . 66
6.2 Future work . . . 68

**PART II: Runtime enforcement for control systems security** **69**

**7 Runtime enforcement for control systems security: a formalisation** **71**
7.1 The model . . . 71
7.1.1 A process calculus representation for PLCs . . . 71
7.1.2 An enforcement mechanism based on edit automata . . . 73
7.2 Use case: the SWaT system . . . 75
7.3 A simple language for controllers' timed properties . . . 78
7.3.1 Syntax and semantics . . . 78
7.3.2 Modularity: from simple to complex properties . . . 79
7.3.3 Local properties . . . 80
Basic properties . . . 80
Compound conditional properties . . . 81
Compound persistent conditional properties . . . 83
Bounded mutual exclusion . . . 84
7.3.4 Global properties . . . 84
7.4 Monitor synthesis . . . 86
7.4.1 Synthesis algorithm . . . 86
7.4.2 Enforcement properties . . . 87
7.5 Summary . . . 89

**8 Our enforcement mechanism at work** **91**
8.1 FPGAs as secure proxies for ICSs . . . 91
8.2 An implementation of the enforcement of the SWaT system . . . 92
8.3 The enforced SWaT system under attack . . . 93
8.4 Summary . . . 99

**9 End of Part II** **101**
9.1 Related work . . . 102
9.2 Future work . . . 105

**10 Overview of published work** **107**

**11 Appendix** **109**

11.1 An introduction to the HMODEST language . . . . . 109

11.2 Proofs of Section 7.4 . . . . . 110

# List of Figures

1.1 A network of compromised PLCs: $y_i$ denote genuine sensor measurements, $y_i^a$ are corrupted sensor measurements, $u_i^a$ corrupted actuator commands, and $c_i^a$ denote corrupted inter-controller communications. . 5
1.2 A network of monitored controllers. . . . . 6

2.1 Main components of control systems in a feedback loop. . . . . 10
2.2 Architecture of modern ICSs. . . . . 11
2.3 Examples of PLCs . . . . . 12
2.4 Examples of HMIs . . . . . 12
2.5 Typical components of a PLC. . . . . 13
2.6 A field network of PLCs. . . . . 13
2.7 Attack vectors in Stuxnet attack on Natanz facility. . . . . 15
2.8 Cyberattack on the Ukrainian power grid. . . . . 16
2.9 Triton malware cripples safety systems in petrochemical plant. . . . . 17
2.10 Attack locations of ICSs. . . . . 18
2.11 Control systems equipped with physics-based attack detection. . . . . 23

3.1 A hybrid automaton modelling a gas-burner. . . . . 29
3.2 Initialized rectangular automaton of a train on a circular track with a gate. 31

4.1 Simulations in MATLAB of *Sys*. . . . . 40
4.2 Implementation in HMODEST of *Sys*. . . . . 41
4.3 Plant() sub-processes. . . . . 42
4.4 Logics() sub-processes. . . . . 44
4.5 Network() process. . . . . 45
4.6 DoS attack to the actuator. . . . . 46
4.7 DoS attack to the sensor. . . . . 47
4.8 Integrity attack to the sensor device. . . . . 48

5.1 Johansson's quadruple-tank water system. . . . . 51
5.2 Implementation in HMODEST of *Sys*. . . . . 53
5.3 Sensors() and Actuators() processes. . . . . 54
5.4 Safety() process. . . . . 55
5.5 Controller of the water level of Tank 1. . . . . 56
5.6 IDS plant estimator. . . . . 56
5.7 IDS control reconfiguration of Tank 1. . . . . 57

5.8 Network() process. . . . . 58
5.9 Integrity attack on the sensor level of Tank 1. . . . . 59
5.10 Red: $\Diamond_{[0,1000]}(\texttt{alarm\_level}_i)$ - Blue: $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_i)$ - Green: $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_i)$ - $i \in \{1,4\}$. . . . . 60
5.11 DoS attack on Pump B to prevent its deactivation. . . . . 61
5.12 Red: $\Diamond_{[0,1000]}(\texttt{alarm\_level}_i)$ - Blue: $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_j)$ - Green: $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_j)$ - $i \in \{1,2\}, j \in \{1,2,3,4\}$. . . . . 61
5.13 DoS on stop commands on Pump B together with a replay attack on measurements of Tank 2. . . . . 62
5.14 Red: $\Diamond_{[0,1000]}(\texttt{alarm\_level}_i)$ - Blue: $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_j)$ - Green: $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_j)$ - $i \in \{1,2\}, j \in \{1,2,3,4\}$. . . . . 63

7.1 A simplified Industrial Water Treatment System. . . . . 76
7.2 Ladder logics of the three PLCs controlling the system in Figure 7.1. . . . 77
7.3 A trace satisfying a persistent conditional property $\mathrm{PCnd}_m(\pi, p)$. . . . . 80
7.4 A trace satisfying a minimum duration property $\mathrm{MinD}(\pi_1, \pi_2, m, n)$, for $m = n = 3$. . . . . 83
7.5 A trace satisfying the aforementioned property for some $m$, $n = m + 4$ and $d = 4$. . . . . 85

8.1 An implementation in Simulink of the plant of the SWaT system. . . . . 92
8.2 Some components of our implementation. . . . . 93
8.3 Tank overflow: Ladder Logic of the first (left) and the second attack (right). 94
8.4 Verilog code of the enforcers of the three properties $e_1, e_2, e_3$. . . . . 94
8.5 Tank overflow: DoS attack on $\mathrm{PLC}_1$ when enforcing $e_1, e_2, e_3$ (up) and $e_1', e_2, e_3$ (down). . . . . 95
8.6 Tank overflow: integrity attack on $\mathrm{PLC}_2$ when enforcing $e_1', e_3$ (up) and $e_1', e_2, e_3$ (down). . . . . 96
8.7 Valve damage: Ladder logic of the first (left) and the second attack (right). 96
8.8 Valve damage: injection attack on $\mathrm{PLC}_1$ in the absence (up) and in the presence (down) of enforcement. . . . . 97
8.9 Valve damage: integrity attack on $\mathrm{PLC}_2$ in the absence (up) and in the presence (down) of enforcement. . . . . 98
8.10 Pump damage: injection attack on $\mathrm{PLC}_3$ in the absence (up) and in the presence (down) of enforcement. . . . . 98

11.1 Master and Slave processes in HMODEST . . . . . 110

# List of Tables

7.1 LTS for controllers. . . . . 73
7.2 LTS for field communications networks of monitored controllers. . . . . 75
7.3 Trace semantics of our regular properties. . . . . 79
7.4 Overview of local properties. . . . . 85
7.5 Monitor synthesis from properties in $\mathbb{P}\mathrm{rop}\mathbb{G}$ and $\mathbb{P}\mathrm{rop}\mathbb{L}$. . . . . 87

11.1 Cross product between two edit automata with alphabet $\mathcal{P}$. . . . . 111

# Chapter 1

# Introduction

*Industrial Control Systems* (ICSs) are physical and engineered systems whose operations are monitored, coordinated, controlled, and integrated by a computing and communication core [129]. They represent the backbone of Critical Infrastructures for safety-critical applications such as electric power distribution, nuclear power production, and water supply. Historically, ICSs relied on proprietary technologies and were implemented as stand-alone networks in physically protected locations. However, in recent years the situation has changed considerably: commodity hardware, software and communication technologies are used to enhance the connectivity of these systems and improve their efficiency.

This computer-based evolution has triggered a dramatic increase in the number of attacks targeting such systems [140, 13, 79]. Some notorious examples of the so called *cyber-physical attacks* are: (i) the *Stuxnet* worm, which reprogrammed Siemens PLCs *to destroy centrifuges* in the nuclear facility of Natanz in Iran [88]; (ii) the *Industroyer* attack on the Ukrainian power grid caused *power outages*; (iii) the recent *TRITON/TRISIS* malware that targeted and *shut down a petrochemical plant* in Saudi Arabia [83]. The gravity of such attacks has been addressed in the 2018 World Economic Forum meeting in Davos.

These attacks have shown that malicious activities from the cyber space targeting ICSs can have *adverse physical consequences*. For instance, attacks on a power grid can cause blackouts, affecting critical infrastructures such as medical systems or water systems or even having a catastrophic effect on the economy and public safety [37], attacks on ground vehicles can cause road traffic accidents [68], attacks on GPS systems can mislead navigation systems and make drivers reach destinations desired by the attackers [151].

Thus, the *physical impact* of cyber-physical attacks puts ICSs security apart from information security [65, 66] and demands for ad-hoc solutions. Paradigmatic examples of such solutions are intrusion detections systems that look into the "physics" of the systems under scrutiny [61] to catch attacks that affect the controlled plant. In particular, these ad-hoc solutions must explicitly address the *timing* and the *duration* of cyber-physical attacks. The timing of the attack is a critical issue as the physical state of

a system changes continuously over time and, as the system evolves, some states might be more vulnerable to attacks than others [85]. For instance, an attack launched when the target state variable reaches a local maximum (or minimum) may have a great impact on the whole system behaviour [86]. Furthermore, concerning the duration of the attack, it may take minutes for a chemical reactor to rupture, hours to heat a tank of water or burn out a motor, and days to destroy centrifuges [88].

Many good surveys on the security of industrial control systems have been recently published (e.g., [61, 63]). They all agree that the main security challenges in ICSs arise when the computation is corrupted either by false sensor signals or by malicious control commands addressed to physical processes. Basically, these two kinds of malicious activities can be achieved by compromising one of the following basic components of ICSs: *physical devices*, i.e., sensors devices and actuator devices; *controllers*, i.e., those cyber-components that are devoted to control physical processes, such as programmable logic controllers (PLCs); *communications network*, connecting controllers with physical devices and other controllers.

## 1.1 Our objectives

The goal of the thesis is to apply formal methodologies to the security of industrial control systems, in particular, we have pursued two lines of research: *(i)* testing the effectiveness of static analysis techniques, i.e., *model checking* [41] and *statistical model checking* [100], when preforming an *impact analysis* of an ICS exposed to cyber-physical attacks and *(ii)* the protection of control systems via a *formal approach* based on *runtime enforcement* to ensure specification compliance in networks of possibly compromised controllers.

**Impact analysis** Risk assessment is a critical step in the implementation of a cyber-defence strategy that finds and prioritizes the vulnerabilities in a system. Prioritization is done based on the likelihood that vulnerabilities are exploited and the *impact* that can occur in the case of an actual attack. Thus, an important part of risk assessment is to reason about the *impact of attacks*. In information systems, attacks have for most practical purposes binary impacts (information was manipulated/eavesdropped, or not). On the other hand, attacks manipulating the sensor or control signals of ICSs can be tuned by the attacker to cause a continuous spectrum in damages [136]. For instance, the impacts of water transmission systems can range from stopping water supply to even compromising water quality [109, 136].

Motivated by the risk assessment application, we propose a *formal approach* based on model checking and statistical model checking to perform an *impact analysis* of ICSs exposed to cyber-physical attacks. The goal of our analysis is twofold: (i) a static evaluation of the *physical impact* of the attacks, in terms of *safety* and possible *deadlocks*; (ii) a static evaluation of the *performance* of intrusion detections systems designed to detect cyber-physical attacks.

In our threat model, we consider *attacks targeting sensors and/or actuators* via either the corresponding physical device or the *communication network* used by the *device*. Furthermore, as we are interested in the impact of attacks, we consider an attacker that has already obtained access to the ICS, and we do not consider how vulnerabilities are exploited, and how the attack is hidden. Instead, we focus on the final objective of the attack to maliciously affect the physical part. This is achieved by manipulating: (i) the sensor measurements, i.e, *reading* and possibly *replacing* the genuine sensor measurements with fake ones, (ii) and/or the controller commands, i.e., *reading*, *dropping* and possibly *replacing* the genuine controller commands with malicious ones.

It is worth noting here that faults and attacks targeting sensors and/or actuators have inherently distinct characteristics [141]. Faults are considered as physical events that affect the system behaviour where simultaneous events do not act in a coordinated way, whereas cyber attacks may be performed over a significant number of attack points and in a coordinated way and might even force the operator to perform erroneous countermeasures [20].

Concerning the formal verification techniques, as said earlier, we consider two widely adopted approaches for the verification of *hybrid systems* (and hence ICSs): *model checking* [41] and *statistical model checking* [100]. Both of them are automated techniques that, given a finite-state model of a system and a formal specification, check whether that property holds for that model. The model describes the possible system behaviours in a mathematically precise manner. On the other hand, the specification is typically expressed in a propositional *temporal logic* and prescribes what the system should do, and what it should not do. In practice, both techniques achieve the *safety verification* of CPSs by solving the *reachability* problem: can an unsafe state be reached by an execution of the system (possibly under attack) starting from a given initial state?

The model checking approach, originally developed by Clarke and Emerson [40], systematically *explores all states* of the system model to determine if the specifications are satisfied by the model. In general, the reachability problem for *hybrid systems* is stubbornly undecidable, although boundaries of decidability have been extensively explored in the past couple of decades [9, 77, 90, 147, 134]. Despite the undecidability results, a number of model checking tools for hybrid systems have been recently proposed [59, 56, 26, 132, 47, 36, 73, 89, 121].

Statistical model checking (SMC) simulates the system model for finitely many executions with the classical *Monte Carlo simulation* [74], and uses hypothesis testing to infer whether the samples provide a statistical evidence for the satisfaction or violation of the specification. As a consequence, SMC does not guarantee a 100% correct analysis, but it allows to bound the error of the analysis, i.e., the maximum probability of *false negatives* and probabilistic *uncertainty*. Furthermore, as it does not consider all reachable states, and therefore it addresses the size barrier of model checking techniques, enabling the analysis of large models such as ICSs.

**Protection of controllers** One of the key components of ICSs are *Programmable Logic Controllers*, better known as PLCs. They control mission-critical electrical hardware such as pumps or centrifuges, effectively serving as a bridge between the cyber and the physical worlds. PLCs have an ad-hoc architecture to execute simple repeating processes known as *scan cycles*. Each scan cycle consists of three phases: (i) reading of sensor measurements of the physical process; (ii) execution of the controller code to compute how the physical process should evolve; (iii) transmission of commands to the actuator devices to govern the physical process as desired.

Due to their sensitive role in controlling industrial processes, successful exploitation of PLCs can have severe consequences on ICSs. In fact, although modern controllers provide security mechanisms to allow only legitimate firmware to be uploaded, the running code can be typically altered by anyone with network or USB access to the controllers (see Figure 1.1). Published scan data shows how thousands of PLCs are directly accessible from the Internet to improve efficiency [128]. Thus, despite their responsibility, controllers are vulnerable to several kinds of attacks, including PLC-Blaster worm [137], Ladder Logic Bombs [67], and PLC PIN Control attacks [3].

As a consequence, extra *trusted hardware components* have been proposed to enhance the security of PLC architectures [111, 114]. For instance, McLaughlin [111] proposed a policy-based *enforcement mechanism* to mediate the actuator commands transmitted by the PLC to the physical plant. Mohan et al. [114] introduced a different architecture, in which every PLC runs under the scrutiny of a *monitor* which looks for deviations with respect to safe behaviours. Both architectures have been validated by means of simulation-based techniques. However, as far as we know, *formal methodologies* have not been used yet to model and formally verify security-oriented architectures.

We propose a *formal approach* based on *runtime enforcement* to ensure specification compliance in networks of controllers possibly compromised by *colluding malware* that may tamper with actuator commands, sensor readings, and inter-controller communications.

*Runtime enforcement* [135, 102, 51] is a powerful verification/validation technique aiming at correcting possibly-incorrect executions of a system-under-scrutiny (SuS). It employs a kind of monitor that acts as a *proxy* between the SuS and the environment interacting with it. At runtime, the monitor *transforms* any incorrect executions exhibited by the SuS into correct ones by either *replacing*, *suppressing* or *inserting* observable actions on behalf of the system. The effectiveness of the enforcement depends on the achievement of the two following general principles [102]:

- *transparency*, *i.e.*, the enforcement must not prevent correct executions of the SuS;
- *soundness*, *i.e.*, incorrect executions of the SuS must be prevented.

Our *goal* is to enforce potentially corrupted controllers using *secure proxies* based on a sub-class of Ligatti's edit automata [102]. These automata will be *synthesised* from enforceable *timed correctness properties* to form networks of *monitored controllers*, as in

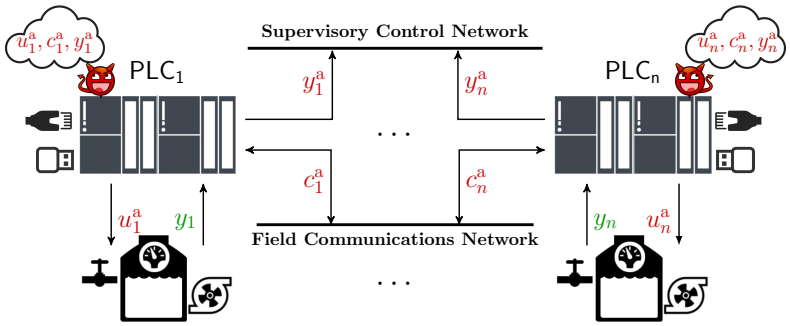



**Figure 1.1:** A network of compromised PLCs: $y_i$ denote genuine sensor measurements, $y_i^{\mathrm{a}}$ are corrupted sensor measurements, $u_i^{\mathrm{a}}$ corrupted actuator commands, and $c_i^{\mathrm{a}}$ denote corrupted inter-controller communications.

Figure 1.2. The proposed enforcement will enjoy both transparency and soundness together with the following features:

- *determinism preservation*, *i.e.*, the enforcement should not introduce *nondeterminism*;
- *deadlock-freedom*, *i.e.*, the enforcement should not introduce deadlocks;
- *divergence-freedom*, *i.e.*, the enforcement should not introduce divergencies;
- *mitigation*, *i.e.*, the enforcer takes over the control of the system when controller is not able to do so;
- *compositionality*, *i.e.*, the enforcement features should hold in networks of controllers.

As expected, when a controller is compromised, these objectives can be achieved only with the introduction of a physically independent *secure proxy*, as advocated in [111, 114], which does not have any Internet or USB access, and which is connected with the monitored controller via *secure channels*. This may seem like we just moved the problem over to securing the proxy. However, this is not the case because the proxy only needs to enforce a *timed correctness property* of the system, while the controller does the whole job of controlling the physical process relying on potentially dangerous communications via the Internet or the USB ports. Thus, any upgrade of the control system will be made to the controller and not to the secure proxy. Of course, by no means runtime reconfigurations of the secure proxy should be allowed.

## 1.2 Contribution

### 1.2.1 An impact analysis for the security of control systems

In order to test the effectiveness of both model checking and statistical model checking, we consider the MODEST TOOLSET [73] which comprises several state-of-the-art analysis backends, in particular it provides (i) a *safety model checker*, called prohver, that relies on

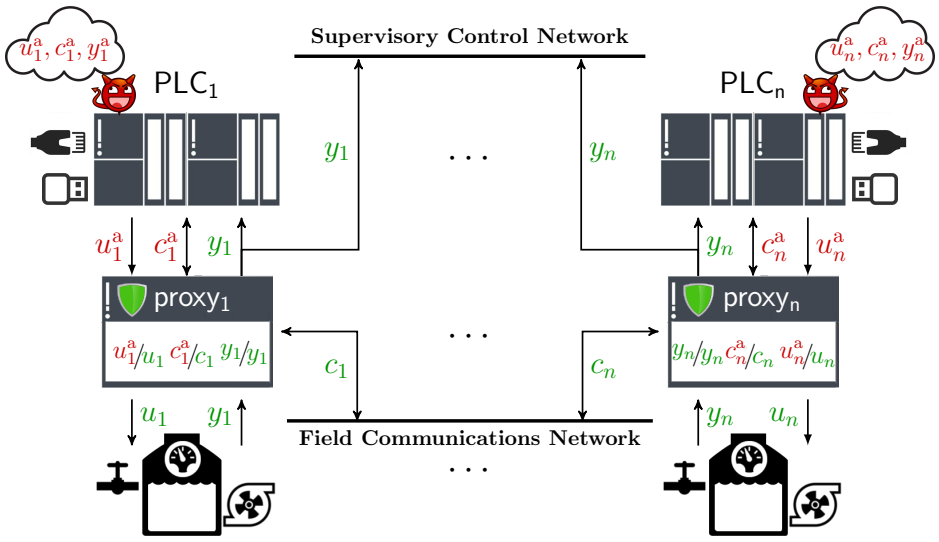



**Figure 1.2:** A network of monitored controllers.

the hybrid solver PHAVer [56] and (ii) a *statistical model checker*, called modes. The toolset relies on a unified modelling language, called HMODEST [72], a process-algebra based language that has an expressive programming language-like syntax to design complex systems.

Concerning model checking, we implement a simple but realistic and nuanced control system which has been proposed by Lanotte et al. [94] to highlight different classes of attacks on sensors and actuators, in a way that is basically independent on the size of the system. We use prohver to analyse three simple but significant cyber-physical attacks targeting sensors and/or actuators of our case study by compromising either the corresponding physical device or the communication network used by the device. The three attacks have already been carefully studied in [94] focussing on the time aspects of the attacks (begin, duration, *etc.*) and the physical impact on the system under attack (deadlock, unsafe behaviour, *etc.*). We then compare its effectiveness in verifying impact of attacks on ICSs, when compared to other state-of-the-art models checkers, such as PRISM [89], UPPAAL [19] and Real-Time Maude [121].

As regards statistical model checking, we analyse a non-trivial *quadruple-tank water system* proposed by Johansson [81]. We equip the original system with an *intrusion detection system* (IDS) that monitors the consistency of both *sensor signals* and *actuator commands*, based on a discrete-time state-space model of the control system under scrutiny. Our IDS is also capable of dynamically reconfiguring the control algorithm in order to *mitigate the physical impact* of detected attacks. We then perform an *impact analysis* of Johansson's tank system and test the effectiveness of modes when doing a security and safety analysis of a significantly larger control system (consisting of 4 physical variables, 2 sensors and 2 actuators) when exposed to three carefully-designed cyber-physical attacks targeting sensors and/or actuators via either the corresponding

physical device or the communication network used by the device.

### 1.2.2 Runtime enforcement for control systems security

We introduce a formal language to specify controller programs. For this very purpose, we resort to *process calculi*, a successful and widespread formal approach in *concurrency theory* for representing complex systems, such as mobile systems [30] and cyber-physical systems [92], and used in many areas, including verification of security protocols [1, 2] and security analysis of *cyber-physical attacks* [95]. Thus, we define a simple timed process calculus, based on Hennessy and Regan's *Timed Process Language* (TPL) [75], for specifying controllers, finite-state edit automata, and networks of communicating monitored controllers.

Then, we define a simple description language to express *timed correctness properties* that should hold upon completion of a finite number of scan cycles of the monitored controller. This will allow us to abstract over controllers implementations, focusing on general properties which may even be shared by completely different controllers. In this regard, we might resort to one of the several logics existing in the literature for monitoring timed concurrent systems, and in particular cyber-physical systems (see, *e.g.*, [17, 58]). However, the peculiar iterative behaviour of controllers convinced us to adopt a simple but expressive sub-class of *regular expressions*, the only properties that under precise conditions can be enforced by finite-state edit automata (see Beauquier et al.'s work [18]). Then we will show how we can express a wide class of interesting correctness properties for controllers in terms of our regular properties.

After defining a formal language to describe controller properties, we provide a *synthesis function* $\langle\!|\ -\ |\!\rangle$ that, given an alphabet $\mathcal{P}$ of observable actions (sensor readings, actuator commands, and inter-controller communications) and a regular property $e$ combining events of $\mathcal{P}$, returns an edit automaton $\langle\!|\, e \,|\!\rangle^{\mathcal{P}}$. In this work we are interested in a deterministic enforcement, thus, we focus on syntactically deterministic regular properties which will give rise to enforcers with a deterministic behaviour. More broadly, the resulting enforcement mechanism will ensure the required features mentioned before: transparency, soundness, determinism preservation, deadlock-freedom, divergence-freedom, mitigation and compositionality.

We propose an implementation of our enforcement mechanism based on *field-programmable gate arrays* (FPGAs) [150], which are good candidates for implementing *secure proxies* as they introduce a negligible overhead in the whole behaviour of the PLCs and are assumed to be secure when the adversary does not have physical access to the device. Furthermore, we propose a non-trivial case study, taken from the context of industrial water treatment systems implemented as follows: (i) the physical plant is simulated in *Simulink* [110]; (ii) the open source PLCs are implemented in *OpenPLC* [11] and executed on Raspberry Pi; (iii) the enforcers run on connected FPGAs. In this setting, we test our enforcement mechanism when injecting the PLCs with 5 different malware, with different goals.

## 1.3 Outline

The thesis is structured as follows. In Chapter 2 we present industrial control systems, programmable logic controllers, their vulnerabilities and associated security measures. In Chapter 3 we provide the technical material about: (i) hybrid model checking, hybrid automata and temporal logics; (ii) statistical model checking and (iii) runtime enforcement and edit automata. The rest of this thesis is divided into two parts. The first part contains *an impact analysis for the security of control systems*. In Chapter 4 we implement in HMODEST an engine system and we put under stress the safety model checker prohver for a security analysis. In Chapter 5 we implement in HMODEST a quadruple-tank water system and test effectiveness of statistical model checking modes. Chapter 6 discusses related and future work. The second part of this thesis contains the contributions of our second objective, *runtime enforcement for control systems security*. In Chapter 7 we present (i) a formal language for monitored controllers, (ii) our case study taken from the context of industrial water treatment systems, (iii) a language of regular properties to express controller behaviours and (iv) the algorithm to synthesise monitors from regular properties, together with formal results. In Chapter 8 we propose an implementation of our enforcement mechanism. Chapter 9 discusses related and future work. Chapter 10 contains an overview of the papers published by the author of this thesis. Finally, the Appendix contains an introduction to the syntactic constructs of the HMODEST language and the technical proofs of the results in Chapter 7.

# Chapter 2

# Industrial control systems and their vulnerabilities

In this chapter we provide an overview of industrial control systems, their vulnerabilities and associated security measures. This chapter has the following structure. In Section 2.1 we introduce the main components of ICSs and take a closer look at the widely used devices for control, programmable logic controllers. In Section 2.2 we present some of the most notorious attacks that have targeted ICSs. In Section 2.3 we analyse cyber-physical attack targetting ICSs in terms of: their stage, the attack locations, the attack motivation, the attacker's goal and knowledge. In Section 2.4 we overview security measures tailored for ICSs.

## 2.1 An introduction to industrial control systems

Industrial control systems (ICSs) integrate *computing* and associated instrumentation to control *physical processes*. More broadly, ICSs are called *cyber-physical systems* (CPSs), which are the emerging applications of embedded computer and communication technologies to a variety of physical domains. CPSs are related to other popular terms such as the *Internet of Things* (IoT) and *Industry 4.0*, but the term CPS does not directly reference either implementation approaches nor particular applications. It focuses instead on the fundamental intellectual problem of conjoining the engineering traditions of the cyber and physical worlds [98]. In the last decade, these systems have increased in number due to both advances in digital electronics and the desire for more information about and control of physical systems. CPSs span many domains, such as: aerospace, automotive, chemical production, civil infrastructure, energy, healthcare, manufacturing and transportation. Some of the benefits of the integration of the computing systems and the the physical world are: optimized productivity, safety, efficiency and predictability.

Finally, due to interaction with the physical world, CPSs present often the following characteristics: real-time, safety-critical and non-reversibility or non premption of

operations. In real-time systems, the time in which computations are performed is important in order to ensure the correctness of the system. The failure of a safety-critical system may have adverse physical consequences, such as serious harm to people, damage to equipment and environmental harm. As regards non-reversibility, once a command affecting the physical part has been executed it is impossible to roll back that operation, whereas in cyber systems roll back operations are typically available.

**Control systems** A control system regulates the behaviour of physical devices via control loops. Such systems range from simple water controllers that regulate pumps to control the water level in a tank to *industrial control systems* that regualate the electricity of nationwide networks of power grids. Control systems were originally designed with analogue sensing and control, which allowed the seamless integration of control signals into a continuous-time physical process. With the advent of microprocessors and computers, discrete-time control systems have emerged, where sensing and control signals have to be sampled at discrete-time. Finally, networks have allowed digital controllers to be further away from the sensors and actuators (e.g., pumps, valves, etc.). Thus, today a control system is typically composed of the (i) physical process, (ii) a communication network that supports (iii) the sensor measurements and actuator data that are exchanged with (iv) controller(s) and supervisor(s), which are the *cyber* components (also called *logics*) of a control system. Figure 2.1 shows the main components of a control systems in a feedback loop, a typical control architecture.

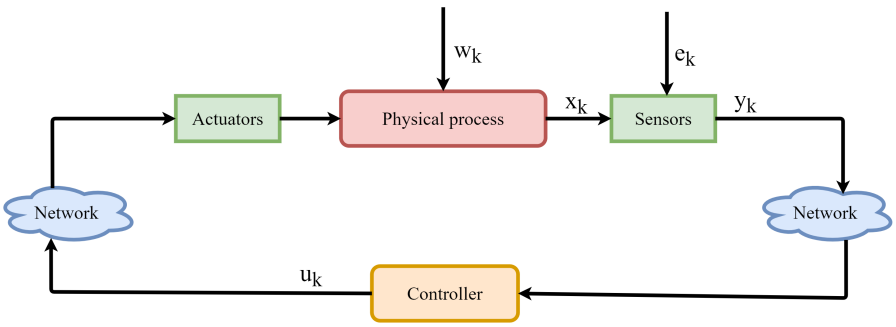



**Figure 2.1:** Main components of control systems in a feedback loop.

The dynamic behaviour of the controlled *physical process* of a control system is often represented by means of a *discrete-time state-space model* consisting of two equations of the form

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ y_k &= Cx_k + e_k \end{aligned}$$

where $x_k \in \mathbb{R}^n$ is the current *(physical) state*, $u_k \in \mathbb{R}^m$ is the *input* (i.e., the control actions implemented through actuators) and $y_k \in \mathbb{R}^p$ is the *output* (i.e., the measurements from the sensors). The *uncertainty* $w_k \in \mathbb{R}^n$ and the *measurement error* $e_k \in \mathbb{R}^p$ represent

perturbation and sensor noise, respectively, and $A$, $B$, and $C$ are matrices modelling the dynamics of the physical system. Here, the *next state* $x_{k+1}$ depends on the current state $x_k$ and the corresponding control actions $u_k$, at the sampling instant $k \in \mathbb{N}$. The state $x_k$ cannot be directly observed: only its measurements $y_k$ can be observed.

**Industrial control systems** ICSs are highly interconnected, interactive and typically span over multiple locations. In critical infrastructure contexts, ICSs are often called Supervisory Control and Data Acquisition (SCADA) systems where they perform vital functions in national critical infrastructures. For instance, the oil and gas industry use integrated SCADA systems to manage refining operations at plant sites, remotely monitor the pressure and flow of gas pipelines, and control the flow and pathways of gas transmission. Water utilities can remotely monitor well levels and control the wells pumps; monitor flows, tank levels, or pressure in storage tanks; monitor pH, turbidity, and chlorine residual; and control the addition of chemicals to the water. Other applications are: electric power distribution, transportation and chemical processing.

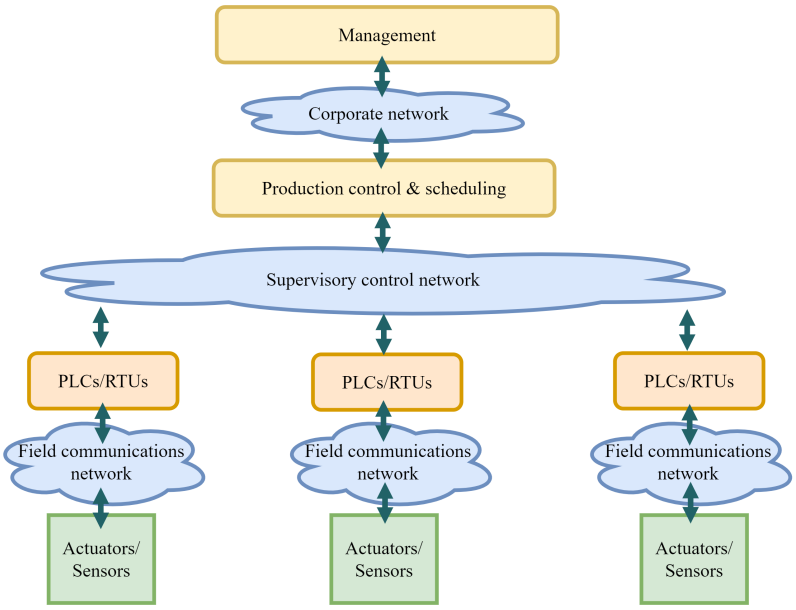



**Figure 2.2:** Architecture of modern ICSs.

As regards the architecture of ICSs, generally, such large distributed control systems may rely on thousands of *field communications networks* of local control loops. The local control loops are seamlessly guided via the *Supervisory control network* by the *production control and scheduling* layer which responds to the upper *management* layer, see Figure 2.2. Typical ICS devices that are devoted to the control of local plant sites are Programmable logic controllers (PLCs) and remote transmission units (RTUs). PLCSs and RTUs transmit acquired telemetry data to the upper layers and execute control logic.

Siemens S7-300

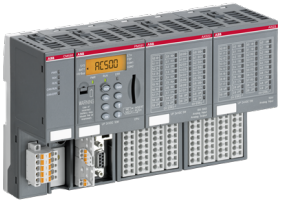
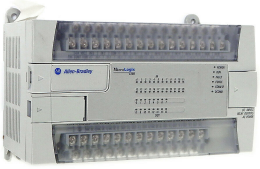


ABB AC500



Allen Bradley Micrologix 1200

**Figure 2.3:** Examples of PLCs

On the other hand, Human-Machine Interfaces (HMIs) devices allow the production control and scheduling layer to communicate with machineries and production plants. Huge amount of complex data are translated into accessible information and displayed on HMIs for the operator that controls the production process. Basically, HMIs are screens that consist of buttons, alarms, reports and trends for monitoring, analysing and controlling the process.


**Figure 2.4:** Examples of HMIs

### 2.1.1 A closer look to programmable logic controller

Programmable logic controllers (PLCs) are one of the primary devices for controlling industrial control systems. PLCs are specialized industrial computers that do not have the same computing power of classical computers and, therefore, they tend to have limited resources. Indeed, PLCs run only on firmware, which is a specific class of software that provides low-level control of device hardware. Furthermore, PLCs are physically hardened, suitable for the harsh production environment. Typically, they have a power supply, one or more communication modules, for communications to ICS servers, HMI or other PLCSs, a control processor, an input module for sensory components and output modules to connect to actuators, see Figue 2.5.

The CPU module of a PLC is its central part, and usually has a microprocessor to handle all the program tasks, which is a programmable memory to store the *user program*, the main program task, and a temporary memory to store the program's data during execution. Thus, a PLC works by continuously scanning the user program, this

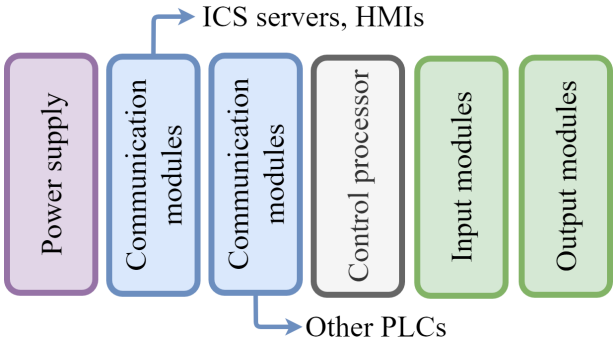



**Figure 2.5:** Typical components of a PLC.

is called scan cycle and consists of three important steps: (i) reading of *sensor measurements* of the physical process, where the state of the actual inputs is copied to a portion of the CPU memory called input image table; (ii) execution of the controller code, to determine how the physical process should change according to both sensor measurements and potential interactions with other controllers, which are also stored in the input image table; (iii) transmission of *commands* to the *actuator devices* to implement the calculated changes, which travel from the output image table to the physical outputs. The scan cycle of a controller must be completed within a specific time, called *maximum cycle limit*, which depends on the controlled physical process; if this time limit is violated the controller stops and throws an exception [137]. As regards the programming languages used to write user programs, according to an international standard set of languages defined in IEC 61131-3, PLCs support five languages: ladder diagram, function block diagram, structured text, instruction list, and sequential function chart [82].

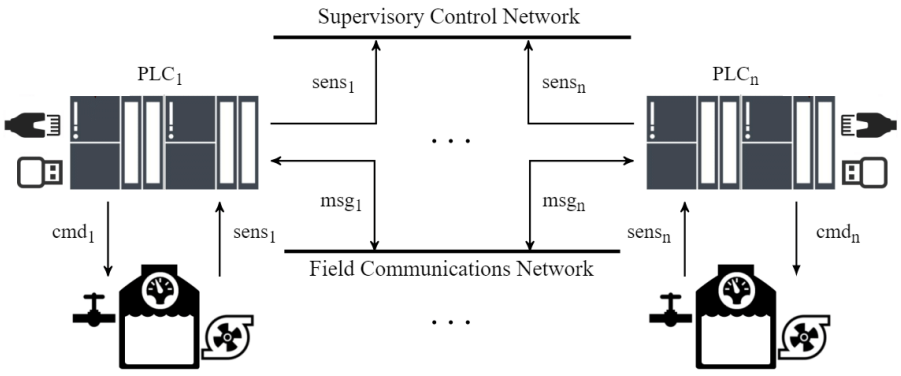



**Figure 2.6:** A field network of PLCs.

Finally, concerning PLCs remote interactions, as said earlier, field communications network allow the interaction between PLCs. On the other hand, the *supervisory control network* allows the PLCs to report various informations, such as the physical state of the plant, to the higher levels, e.g. HMIs (see Figure 2.6). Typical industrial network protocols are: DNP3, Modbus/TCP, EtherNet/IP, PROFINET, ICCP, and IEC 104 [118].

## 2.2 High profile attacks targeting ICSs

The modernization of control systems over the past decade has raised many concerns about the vulnerabilities in control systems to security attacks. In particular, modern ICSs rely on reprogrammable devices which communicate remotely even via the Internet which makes them exposed to new types of threats and increases the possibility that an ICS could be compromised with cyber-attacks. These attacks can induce the failure of ICSs and cause economic losses or even worse, due to their safety-critical nature, their failure can cause irreparable harm to the physical system being controlled, contaminate ecological environment, and harm the people who depend on it.

In what follows we present attacks that were specifically designed for ICSs and that made it to the news due to their sophistication and their disruptive physical impact. In particular we present: the attack that targeted the SCADA system in Australia, the Stuxnet worm, the attack that damaged a blast furnace in Germany, the attacks that disrupted the Ukrainian power grid and the Triton malware that has affected petrochemical plants in Saudi Arabia.

**Attack on an Australian SCADA system - 2007** The sewage treatment facility in Queensland, Australia, was attacked and its SCADA system was manipulated to release raw sewage into local rivers for three months [136]. This was the first publicly reported attack on a SCADA system, where the attacker was a contractor who wanted to be hired for a permanent position maintaining the system. He used commercially available radios and stolen SCADA software to make his laptop appear to be a pumping station. During a three-month period the attacker released more than 750,000 gallons of untreated sewage water into parks, rivers, and hotel grounds, causing loss of marine life, jeopardising public health, and costing more than $200,000 in clean-up and monitoring costs.

**Stuxnet - 2013** The *Stuxnet* worm reprogrammed PLCs to damage nuclear centrifuges in the nuclear facility of Natanz in Iran [88]. This attack, which is paradigmatic example of a weaponized malware, was reportedly state-sponsored and showed an unprecedented sophistication for cyber attacks. Stuxnet was designed to infect as many computer as possible, but on many computers had no effect. When it found its intended target, in this case computers in the nuclear facility of Natanz, used multiple zero day exploits, bypassed intrusion detection systems, disguised itself as legitimate software and then covered its tracks by removing trace files from systems if they were no longer needed or considered incompatible. The first attack vector propagated inside the air-gapped nuclear facility via infected USB drives. Once inside, it took over the PLCs. The control logic of the malware recorded and replayed the sensor values of the rotor vibration and pressure to hide the ongoing attack and, at same time, shut off exhaust valves of the centrifuges leading to pressure build-up and ultimately damaging the centrifuges. The second attack vector tampered with the Centrifuge Drive System (CDS), which controlled the rotor speeds of the centrifuge system. Now Stuxnet

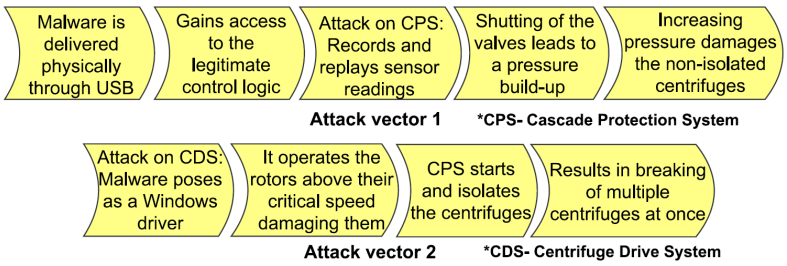



**Figure 2.7:** Attack vectors in Stuxnet attack on Natanz facility.

used copies of stolen digital certificates and posed as a legitimate driver software for the Windows Operating System. The centrifuges were damaged by increasing the rotor speed beyond a critical value where the harmonics (distortions in power systems) were triggered, these harmonics damaged the rotor walls. Furthermore, the centrifuges used an in-built protection system called Cascade Protection System which isolated the troubled centrifuge tubes. Thus, once the rotors were damaged, the Cascade Protection System isolated the damaged centrifuges. Beyond a certain number of isolated centrifuges, the system would shut off the valves of the remaining centrifuges. This would induce an increase in pressure to the non-isolated centrifuges and ultimately these centrifuges would break as well [8].

**Attack on a German steel mill - 2014** In 2014 an attacker gained access to the control systems via the steelworks business network and forced the shut down of a blast furnace in a German steelworks [99]. The attacker used "spear phishing" techniques to steal logins and gain access to the mill's control systems. The attacker then caused the failure of some parts of the plant which prevented the normal shutdown of a blast furnace and, ultimately, the unscheduled shutdown of the furnace caused the damage.

**Attacks on the Ukrainian power grid - 2015/2016** Other high-profile attacks against ICS (and most likely state-sponsored) occurred in December 2015 and 2016 on the Ukrainian power grid. These attacks caused power outages during the winter season by disrupting three energy distribution companies. These attacks included, spear phishing attacks to infiltrate the corporate network, disabling/destroying SCADA and IT infrastructure, destruction of files stored on servers and Telephony Denial of Service (TDoS) attack that flooded the call centres to block the real customer calls from getting through. Finally, the attacks in 2016 leveraged the *Industroyer* malware [37] to automate the 2015 attack, in those attacks no human intervention was required to remotely operate the human-machine interfaces.

**Triton - 2017** The recent *Triton* malware targeted a petrochemical plant in Saudi Arabia [83]. The attack goal was to disable safety instrumented systems to cause physical damage to the plant. However, it has never reached its intended goals as the attack was detected during the injection of the malware into the controller memory, a supposedly

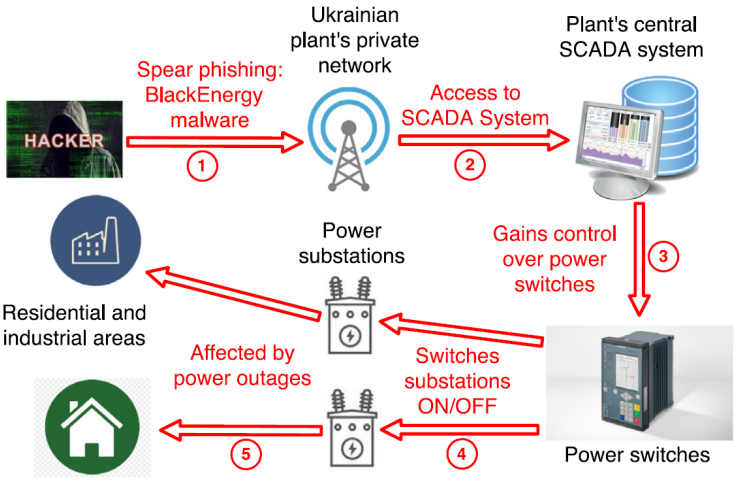



**Figure 2.8:** Cyberattack on the Ukrainian power grid.

early stage ot the attack. In that initial phase of the malware the devices went into the fail-safe mode, causing operations to pause at multiple facilities and hence triggering a shutdown.

These are the most remarkable attacks, nonetheless, there has been a dramatic increase in the number of attacks [79] with the introduction of *Industry 4.0*, i.e., the growing connectivity and integration of these systems. Furthermore, Stuxnet and Industroyer stand out as they show that the motivation and capability exists for creating computer attacks capable to achieve military goals and indicate an arms race in state-sponsored cyber attacks.

## 2.3 Anatomy of cyber-physical attacks targeting ICSs

In this section, we take a closer look at cyber-physical attacks. In particular, we consider these kind of attacks in the context of ICSs and we examine: (i) the *stages* of the attack, (ii) the *attack locations* of an ICS, i.e., the parts of the system that may be manipulated during an attack, (iii) the *motivation* of the attacker, (iv) the attacker's *goal* and (v) the attacker's *knowledge*, that is needed to successfully execute an attack.

In general, we can define cyber-physical attacks as follows:

**Definition 1.** *Cyber attacks that (a) require an expert knowledge in the domain of the physical components, (b) disrupt the normal operations of control systems through the cyber-space and (c) bring the plant into a unsafe/incorrect state possibly with catastrophic physical consequences.*

**Attack stages** There are several stages that lead to a successful attack on a ICS, an attacker needs the means to manipulate it and a deep knowledge of both the physical and

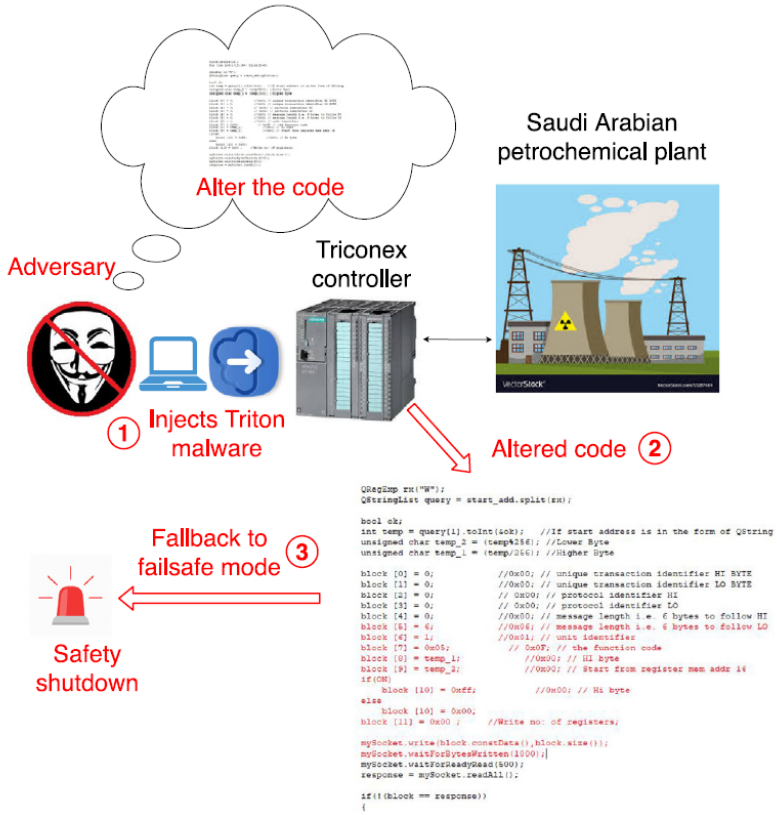



**Figure 2.9:** Triton malware cripples safety systems in petrochemical plant.

the cyber part. Gollmann et al. [66] identified the following stages: access, discovery, control, damage, clean-up.

- *access*: the attacker finds a way inside the ICS network. Potential ways in are the vulnerable links across the field, control and the corporate networks. Furthermore, social engineering technique can exploit the employees to provide unauthorized access to the attacker, for instance, via spear phishing techniques. This is the stage that most resembles traditional IT hacking;
- *discovery*: refers to discovering information about a plant to reconstruct its layout and how it carries out its functions. Without a detailed knowledge it's likely that the attacker cannot achieve more than nuisance. Blindly trying to destroy a process will probably only result in exercising the emergency shutdown logic. The attacker may study general information on the dynamics of the physical processes of interest. Furthermore, engineering documentation of the plant can be essential as plants tend to be highly proprietary, and, even if there are engineering standards, ad hoc choices might be involved in the design of the plant;
- *control*: the attacker tries to discover the dynamic behaviour of the plant in terms of simple physical equations, such as differential equations, to understand what each actuator does and what side effects are possible. With the knowledge of the

plant the attack has a better chance to reach its goals. Indeed the transitioning of the process from one state to another is in most cases not instantaneous and adjusting one part of the process for malicious purposes may have side effects on other parts of the process an trigger alarms;

- *damage*: the attacker has gained access to the ICS, has a rough idea of the plant and the control architecture and is ready to take action. There are several choices: *equipment damage*, e.g., bouncing the pumps of the floor until they break, *production damage*, e.g. altering the production rate and *compliance violation*, e.g., exceeding pollution limits;
- *clean-up*: when an ICS attack is successful, a piece of equipment will be damaged or the plant has suddenly operated outside the normal operational ranges. Thus, someone will be sent to investigate. The clean-up phase is about creating a forensic footprint for investigators by manipulating the process and the logs in such a way that the analyst draws the wrong conclusions. The goal is to get the attack blamed on operator error or equipment failure instead of a cyber event.

**Attack locations** In the damage stage the attacker has gained *access* to the system and has a rough idea of the plant and the control architecture, thus, is ready to affect the targeted ICSs. To do so, the attacker can manipulate: the physical devices, the communications network and the controllers that are devoted to control physical processes as depicted in Figure 2.10:

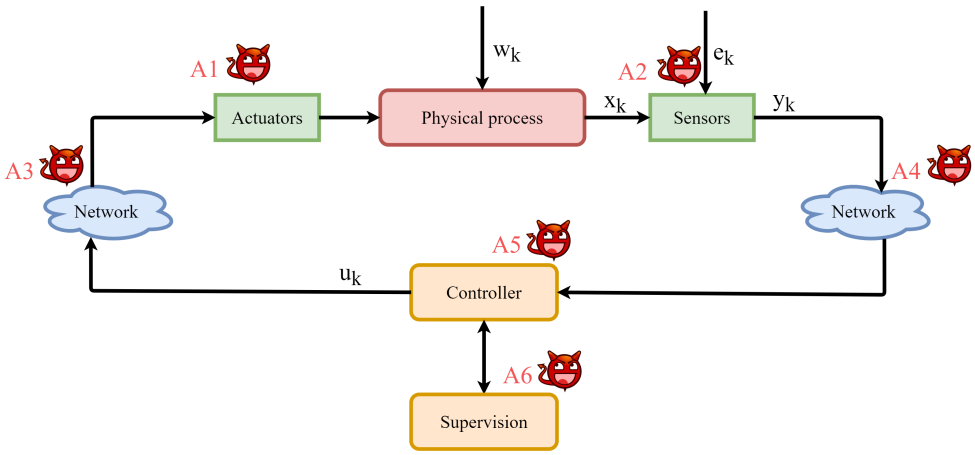



**Figure 2.10:** Attack locations of ICSs.

- Physical devices, i.e., actuators and sensors: (A1) a compromised actuator executes a control action that is different to what the controller intended and (A2) a compromised sensor can *inject* into the system a *fake measurements* which can induce the controller to act maliciously;
- Communications network, at different levels: at the level of field communications network, connecting controllers with physical devices and other controllers,

(A3) the attacker *delays* or *drops* control commands to cause a denial of control to the system or (A4) she can delay or drop sensor measurements intended to the controller which loses observability of the system; (A6) at the level supervisory control network, connecting SCADA and HMI systems to controllers;

- Controllers: (A5) a compromised controller can send incorrect signals to the actuator; (A6) a compromised supervisor can send malicious configurations changes to the controller.

**Attacker's motivation** Attackers that target standard information systems can be motivated by monetary profits and steal personal/secrete information, mine crypto currencies [155], ask for a ransom to unlock data [112], run bot servers [14]. On the other hand, attackers that target specifically ICSs and the disruption of their physical process may play with the system's physical parameters out of curiosity [69]. Disgruntled employees with an expert knowledge on the ICSs may seek revenge [136]. Finally, groups of attackers with political purposes, such as hacktivists or state-sponsored attackers, may launch more sophisticated attacks [88, 37, 83].

**Attacker's goal** Once the attacker has gained access to the system she may have three objectives according to [65]: (i) *equipment damage*, such as overstress of equipment, to reduce their expected life cycle, or violation of safety limits [88]; (ii) *production damage*, the attacker can compromise the product quality, the product rate or the operating costs [37]; (iii) *compliance violations*, such as increasing environmental pollution [136], so that the target plant could be fined, and recurrent offences could lead to plant shutdown.

**Attacker's knowledge** Successful attacks that target ICSs require a deep understanding of both the cyber components and the physical processes involved. Attacker compromising the cyber components of ICSs, must be aware of their ad hoc architecture of control hardware and software. For instance, as mentioned in the Introduction, PLCs execute simple repeating processes known as *scan cycles*. Any scan cycle of a PLC must be completed within a *maximum cycle limit* which depends on the controlled physical process; if this time limit is violated the PLC stops and throws an exception [137]. Thus, a malware that aims to take control of the plant has no interest in delaying the scan cycle and risking the violation of the maximum cycle limit whose consequence would be the immediate controller shutdown.

With regards the knowledge of the physical process, the attacker must take in to account the timing and the duration of the attack in order for her malicious actions to have an physical impact. *Timing* is a critical factor, as the physical state of a system changes continuously over time and, as the system evolves, some states might be more vulnerable than others. For instance, an attack launched when the target state variable reaches a local maximum (or minimum) may have a great impact, whereas the system might be able to tolerate the same attack if launched when that variable is far from its

local maximum or minimum. Furthermore, the *duration* is another critical factor, as it may take minutes for a chemical reactor to rupture, hours to heat a tank of water or to burn a motor, and days to destroy a centrifuges. Finally, *stealthiness* is a necessary condition to complete a malicious activity on a ICSs which may require a certain amount of time. Therefore the attacker must carefully choose her actions to avoid to be detected and a the same time reach her objectives.

## 2.4 Security measures for ICSs

As ICSs operate by integrating the cyber space with the physical plant, cyber-attacks targeting ICSs can cause tangible effects in the physical world. Consequently, as said in the Introduction, this is a major concern and puts cyber-physical systems security apart from information security. Specifically, standard security measures focused traditionally on the protection of information. Such security measures do not consider how attackers that manage to bypass some basic security mechanisms can affect the control algorithms and ultimately, the physical world [31].

In general, information security measures try to address at least one of the three CIA (confidentiality, integrity and availability) security goals. Let us analyse how these standard security goals are related to ICSs.

- *confidentiality* requires unauthorized persons not to have access to information related to the control system. The confidentiality of critical information such as passwords, encryption keys are vital for the protection of the ICS. Furthermore, the confidentiality of the physical state of the plant is important as well, some physical states might be more vulnerable than others;
- *integrity* requires data generated, transmitted, displayed, stored within a control system being genuine and intact without unauthorized intervention, including both its content, which may also include the header for its source, destination and time information besides the payload itself;
- *availability* requires that any device within the system should be ready for use when is needed. As most of controlled processes are critical and continuous in nature, such as power grids, unexpected outages of systems that control industrial processes are not acceptable.

These security goals are not mutually exclusive, for instance, an attacker that breaches the integrity can change the control commands to induce a device malfunction and ultimately affect the availability of the system. In addition, ICSs bring new security goals such as: *timeliness* and *graceful degradation* [46, 152]:

- *timeliness* expresses the time-criticality of control systems, i.e., a command from the controller to the actuator should be executed in real-time by the latter, and the timeliness of any related data, such as measurements, being delivered in its designated time period, as the data is only valid in that time period;

- *graceful degradation* requires the system being capable of keeping the attack impact local and withholding tinted data flow within tinted region without further escalating into a full scale, full system cascading event.

Furthermore, unlike information systems where it is vital to protect the central server and not the edge client. In process control, an edge device, such as a PLC, is not necessarily merited less important than a central host such as data historian server. A compromised PLC can damage significant parts of the plant [88].

In what follows we overview the challenges in implementing standard security best practices and also the stat-of-the-art proposals in implementing and designing security measures for ICSs.

**Security best practices** As a first step in the protection of ICSs, security engineers must follow the security best practices of classical IT systems, such as: follow a secure development lifecycle to minimise software vulnerabilities, implement access control mechanisms, provide strong cryptographic protections along with a secure key management system, use state of the art intrusion detection software, etc. Several standards have been developed [138, 25, 119, 116, 29], for instance, the U.S. National Institute of Standards and Technology (NIST) has a Guide to ICS Security [138], a guideline to smart grid security [119] and a guideline for IoT security and privacy [25].


However, there are several challenges in implementing security best practices for ICSs. First of all, ICSs rely on embedded device with limited resource that often can't implement basic security mechanisms such as cryptographic functions. Furthermore, ICS plants need to operate 24/7, thus, cannot be stopped to update vulnerable components, such as old legacy devices. In addition, the life cycle of ICSs is larger than classical IT systems, for instance industrial asset owners expect their control systems to last for at least 25 years [12]. As a consequence, even if these devices were deployed with security mechanisms at the time, new vulnerabilities will eventually emerge. Finally, updating the security of devices is challenging as devices tend to be certified and any changes in software or operational practices must be followed by an extensive safety revision or re-certification [43]. Thus, to secure ICSs, it is necessary (i) to design systems that support continuous security updates and (ii) to retrofit security solutions for existing legacy systems.

### 2.4.1 State of the art

In what follows we present an overview of what has been proposed to secure ICSs beyond standard security best practices which do not address properly the security challenges of ICSs because they do not consider the physical component of ICSs. In particular, we discuss: (i) techniques for the *detection* of attacks, (ii) the *mitigation* of adverse effects of attacks and (iii) *risk assessment*.

**Detecting attacks** In the last years, security researchers have proposed several intrusion detection algorithms for ICSs. A number of proposed algorithms, very much like classical IT system, monitor the network to identify anomalies in the information exchanged between devices. Unlike classical IT networks, ICSs networks tend to have static topologies, regular traffic patterns, and a limited number of applications and protocols running on them. Consequently, anomaly detection algorithms that use models of the network traffic and white listing access control are easier to design [38, 144]. However, such anomaly detection algorithms may not spot attackers that tamper with the sensor and the control values. In particular, an attacker that has obtained control of a *sensor*, an *actuator*, or a *PLC* (or their corresponding communication network) can insert legitimate commands or measurements between authorized systems and in full compliance with protocol specification and can bring an ICS to perform a function that is outside the owner's intended purposes. Hence, these attacks may not affect the traffic flow but may cause sever damage to the plant [71].

To detect compromised controllers tampering with actuation commands, it has been proposed to rely on some form of control redundancy to verify that the control actions are actually those intended [62, 111, 114]. This is necessary because the attacker can use the controller to send dangerous control signals to the actuator, while hiding the attack by sending false sensor measurements to the supervisory levels [88, 62]. On the other hand, to detect compromised sensors and/or actuators or the communication network used by such devices, a growing line of work has proposed to use models of the physical evolution of the controlled plant [61], called *Physics-Based Attack Detection*. Specifically, the physical evolution of the state of a system follows immutable laws of nature, for example, if the intake valve is opened, then the water level in the tank should rise. Therefore, the physical properties can be used to create time-series models that can confirm that the control commands sent to the field were executed correctly and that the information coming from sensors is consistent with the expected behaviour of the system. Indeed, attackers may hide specific information technology methods used to exploit the control system, but they cannot hide their final goal: the need to cause a physical adverse effect.

This approach is inspired by the field of fault diagnosis, which has been using for a long time physical models to detect defective devices [123]. The main difference is that fault diagnosis techniques do not usually consider strategic adversaries, and if used

for security purposes, they might force the operator to perform erroneous countermeasures [20].

In the past couple of years, an increasing number of publications appearing in security conferences have proposed physics-based attack detection algorithms addressing water control systems [70], power grids [104], boilers in power plants [148], chemical process control [31] and many others [61]. Here it is worth showing the architecture of control systems enhanced with a detection component that looks into the "physics" of the system, see Figure 2.11. There, the detection system receives as inputs the sensor measurements $y_k$ from the physical system and the control commands $u_k$ sent to the physical system and then uses them to identify any suspicious activity within the sensors and/or actuators.

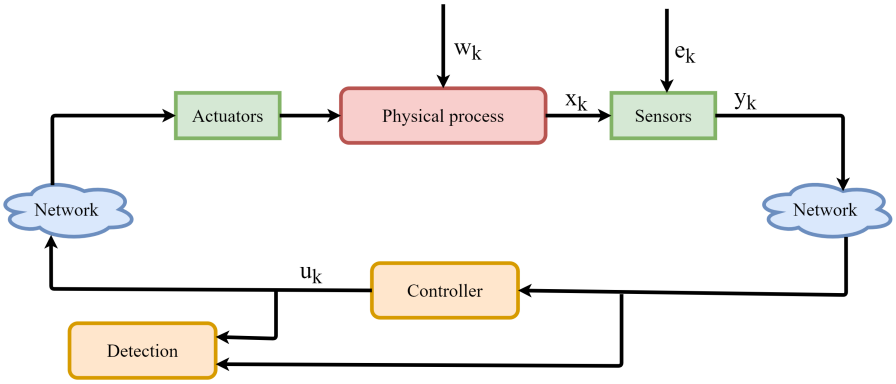



**Figure 2.11:** Control systems equipped with physics-based attack detection.

**Mitigating attacks** Once a threat is detected mitigation mechanisms must thwart the ongoing attack. In classical IT systems web servers send CAPTCHAs to the client whenever they find that connections resemble bot connections, firewalls drop connections that conform to their rules, the execution of anomalous processes can be slowed down by intrusion detection systems. Ultimately, if an attack cannot be stopped, the system can be unplugged. Shutting down a compromised ICS might not be as feasible because, for instance, equipment could be already irreparably damaged and there might not be any safety procedure that can prevent the catastrophic effects of the attack [99, 88].

Control engineers have dealt for long with the protection of the system against random and/or independent faults but not with faults induced by a strategic attacker [123]. As a consequence, researchers have advocated for ad-hoc *reactive response* mechanisms against cyber-physical attacks that protect ICSs when an attack has been detected [153]. Specifically, the goal is to design systems where even if attackers manage to bypass some basic security mechanisms, they will still face several control-specific security devices that will minimize the damage done to the plant [31].

Concerning the mitigation of attacks, in this thesis we focus on compromised controllers. The interested reader can find more material regarding the remaining compromisable locations in [42, 61]. Regarding the security of controllers, and in particular PLCs, as said in the Introduction, controllers are already vulnerable to a number of attacks [137, 67, 3] and thousands of them are directly accessible from the Internet to improve efficiency [128]. Thus, to protect controller in ICSs, an extra *trusted hardware component* has been proposed, that acts as *proxy* between the PLC and the environment interacting with it [111, 114, 124]. In particular, the proxy should be physically independent, not have any Internet or USB access, and connected with the monitored controller via *secure channels*.

The proxy only needs to keep the system *safe*, while the controller does the whole job of controlling the physical process relying on potentially dangerous communications via the Internet or the USB ports. For instance, McLaughlin [111] proposed the introduction of an enforcement mechanism, called $C^2$, which acts according to regular expression-like *security policy* and the current state of the plant. $C^2$ may (a) *drop* a PLC request, (b) *retry* a PLC request once, (c) *approximate* a request on a continuous device, (d) *notify* the PLC that a previous operation was denied. Mohan et al. [114] proposed their *Secure System Simplex Architecture* (S3A) to check not just the physical state of the plant but also the *cyber state* of the PLCs of the system, such as real-time constraints, memory usage, and communication patterns. A violation of the physical model or the computational model will lead to the transfer of control a secure safety controller. Pearce et al. [124] proposed an enforcer that corrects both inputs and outputs according to a policy represented as a *value discrete timed automata* (VDTA) which models safe execution of the the system under scrutiny. The enforcement policies are the following: (i) random corrections, where a value is chosen randomly from a list of input/output values that satisfy the VDTA; (ii) minimum distance corrections, where a value is chosen such that satisfies the VDTA and the value also has the minimum binary distance compared to the current value; (iii) ad-hoc corrections, which are values selected by the engineers out of the list of possible safe values, to ensure practical runtime enforcement.

**Risk assessment** Risk assessment finds and prioritizes the vulnerabilities in a system. An important part of risk assessment is to reason about the *impact of attacks* which allows to prioritize vulnerabilities. In information systems, vulnerabilities are commonly found and exploited via practical analyses. For instance, phishing simulation tools identify risky employee behaviour, professional hackers find and exploit security vulnerabilities in the network or web applications. Practical analyses that identify and actively exploit weaknesses in ICSs often are not feasible, as pointed out by [15]. Indeed, real-world ICSs are often not open to security researchers, furthermore, actively attacking real ICSs could impact system availability and the attack could even cause costly damage [15]. Consequently, to allow security research and study the impact of cyber-physical attacks targeting ICSs, a number of approaches have been proposed [15,

139, 143, 113, 109, 122]. In practice, these works aim to provide the tools for the analysis of a comprehensive list of cyber-physical threat scenarios across a wide range of attack vectors throughout an ICS by somehow emulating (i) the cyber layer information flow and (ii) the physical processes of ICSs and, last but not least, (iii) cyber-physical attacks.

Simulation-based tools have been first proposed [15, 139], which simulate typical ICS components, such as programmable logic controllers, industrial networks and the physical plant, and simulate to custom attacks that can alter the values and the normal flow of commands and sensor values. For instance, Antonioli et al. [15] proposed MiniCPS, based on Mininet [97], the lightweight real-time network emulator. Taormina *et al.* [139] modelled the interaction between the physical and cyber layer of water distribution systems in epanetCPA to study the hydraulic response of water networks during attacks.

Testbeds have also been proposed for research purposes in the field of cyber security of ICSs [109, 122]. Testbeds are small but representative replicas of real-world systems. For instance, Mathur et al. [109] designed SWat, a 6-stage water treatment process which is inspired by large and modern water treatment plants found in large cities. SWaT supports man-in-the-middle attacks between any two parties (e.g. two PLCs) that can eavesdrop on all exchanged sensor and command data and re-write sensor or command values.

Finally, recent works [143, 113, 94] have proposed to perform impact analyses of ICSs equipped with control-specific protection mechanism, such as IDSs that leverage the physics of the plant. Such analyses address the specific features of cyber-physical attacks. For instance, an attacker may have knowledge of the detection mechanism and may leverage this knowledge to evade the detection scheme [143]. Indeed, attackers who want to remain undetected can attempt to hide their manipulations by closely following the system's intended behaviour, while injecting just enough false information to reach their goals. Thus, an impact analysis of an ICS equipped with an intrusion detection system (that leverages the physics of the plant) checks whether an attacker can inflict large damage to the ICS while remaining *stealthy*, i.e., undetected by the intrusion detection system [113].

# Chapter 3

# Technical background: formal verification methodologies

In this chapter we provide an overview of the technical material required to understand our contributions. This chapter has the following structure. In Section 3.1 we introduce model checking for hybrid systems considering: hybrid automata, (ii) the metric temporal logic (MTL) and (iii) the stat-of-the-art tools for hybrid model checking. In Section 3.2 we overview the inner workings of statistical model checking. In Section 3.3 we present the key concepts of the formal approach in runtime enforcement and we present Ligatti et al.'s edit automata.

## 3.1 Model checking of hybrid systems

Industrial control systems and more generally cyber-physical systems are classified as *hybrid systems*, that is, systems that can both flow continuously and jump discretely. A widely adopted mathematical model for hybrid systems is the *hybrid automaton*, which represents discrete components using finite estate machines and continuous components using real-numbered variables [41]. On the other hand, formal specification for hybrid system are based on temporal logics coupled with continuous time constraints, such as *metric temporal logic* (MTL) .

### 3.1.1 Hybrid automata

Here we use the widely adopted definition hybrid automata of [41], in order to ensure that this work is somewhat self contained we provide a fairly detailed presentation of the syntax and semantics of hybrid automata. The reader who is already familiar with such notions can skip the following subsection.

Before presenting hybrid automata, we need to introduce *terms* and *constraints*. Let $X = \{x_1, \ldots, x_n\}$ be a set of of real-numbered variables. Given a valuation function $v : X \to \mathbb{R}$ and $Y \subseteq X$, the valuation function $v_{|Y} : Y \to \mathbb{R}$ is defined as $v_{|Y}(x) = v(x)$ for every $x \in Y$.

A *term* over a finite set of variables $X = \{x_1, \ldots, x_n\}$ is an expression of the form $f(x_1, \ldots, x_n)$. Given $y = f(x_1, \ldots, x_n)$ and a valuation $v$ over $X$, $[\![y]\!]_v$ denotes the real number obtained by evaluating the term at $v$. Two well-known types of functions are worth mentioning here: *affine functions*, i.e., if $f(x_1, \ldots, x_n) = a_1 x_1 + \ldots a_n x_n + b$, for $a_i, b \in \mathbb{R}$ and $1 \leq i \leq n$, and *linear functions*, i.e., affine functions where the parameter $b = 0$. We denote $\text{Term}(X)$ as the set of all terms over the variable $X$. A *constraint* over $X$ is a finite formula $\phi$ defined according to the following grammar:

$$\phi \quad ::= \quad \theta \bowtie c \quad \mid \quad \phi \wedge \phi \quad \mid \quad \phi \vee \phi$$

where $\theta \in \text{Term}(X)$, $\bowtie \in \{<, \leq, =, >, \geq\}$ and $c \in \mathbb{R}$. Finally, $\text{Constr}(X)$ is the class of constraints over the set variables $X$.

As regards the evaluation of constraints, given a valuation $v : X \to \mathbb{R}$ and a constraint $\phi \in \text{Constr}(X)$, we write $v \models \phi$ and say that $v$ satisfies $\phi$, which is defined inductively as follows:

$$\begin{array}{lll} v & \models & \theta \bowtie c \text{ if } [\![\theta]\!]_v \bowtie c \\ v & \models & \phi_1 \wedge \phi_2 \text{ if } v \models \phi_1 \text{ and } v \models \phi_2 \\ v & \models & \phi_1 \vee \phi_2 \text{ if } v \models \phi_1 \text{ or } v \models \phi_2 \end{array}$$

Everything is in place to present the formal definition hybrid automata.

**Syntax** A *hybrid automaton* is a tuple $H = (\text{Loc}, \text{Lab}, \text{Edg}, X, \text{Init}, \text{Inv}, \text{Flow}, \text{Final})$ where:

- $\text{Loc} = \{l_1, \ldots, l_m\}$ is a finite set of locations;
- Lab is finite set of labels, including the silent label $\tau$;
- $\text{Edg} \subseteq \text{Loc} \times \text{Lab} \times \text{Loc}$ is a finite set of edges;
- $X = \{x_1, \ldots, x_n\}$ is a set of real-numbered variables. The number $n$ is called the dimension of $H$. Furthermore, $\dot{X}$ denotes the set $\{\dot{x}_1, \ldots, \dot{x}_n\}$ which represent first derivatives during continuous change, whereas $X'$ for the set $\{x'_1, \ldots, x'_n\}$ of primed variables which represent values at the conclusion of discrete change;
- $\text{Init} : \text{Loc} \to \text{Constr}(X)$ returns the initial condition $\text{Init}(l)$ of location $l$. The automaton can start in $l$ with initial valuation $v \in [\![\text{Init}(l)]\!]$;
- $\text{Inv} : \text{Loc} \to \text{Constr}(X)$ returns the evolution domain restriction $\text{Inv}(l)$ (also called the invariant) of location $l$. The automaton can stay in $l$ as long as the values $v$ of its variables lie in $[\![\text{Inv}(l)]\!]$;
- $\text{Flow} : \text{Loc} \to \text{Constr}(X \cup \dot{X})$ is the flow constraint, which constrains the evolution of the variables in each location. Basically, the flow constraint defines the continuos dynamics of a hybrid automaton, i.e., the values of the variables $X$ change according to the $\text{Flow}(l)$ of the current location $l$ as time moves forward. Formally, in a location l, if the valuation of the variables is $v_0$ at time $t = 0$, then

at time $t \geq 0$, the value of the variables if $\phi(t)$ where $\phi : \mathbb{R} \to \mathbb{R}^X$ is such that the flow relation $\text{Flow}(l)(\phi(t), \dot{\phi}(t))$ holds for the flow $\phi(t)$ and its time derivative $\dot{\phi}(t)$ and $\phi(0) = v_0$.

- $\text{Jump} : \text{Edg} \to \text{Constr}(X \cup X')$ returns the jump condition $\text{Jump}(e)$ of edge $e$. Jump conditions are often conjunctions of a *guard* and a *reset constraint*. Guards are the constraints defined over variables in $X$, and *resets* are the constraints defined over the variables in $X'$ (in terms of variables in $X$, respectively).
- $\text{Final} : \text{Loc} \to \text{Constr}(X)$ gives the final condition $\text{Final}(l)$ of location $l$. Depending on the analysis question at hand, final conditions can either specify the unsafe states of the system or the desired states of the system.

In practice, a hybrid automaton is a graphs whose edges represent discrete transitions and whose vertices represent continuous activities. Note that the labels on the edges can be used to synchronize hybrid automata in a compositional design. For simplicity we omit the formal definition of synchronized hybrid automata, the interested reader may find the definition of composition of hybrid automata in [41]. Having presented the syntax of hybrid automata, in Figure 3.1 we show an automaton modelling a single gas-burner that is shared for heating alternatively two water tanks.

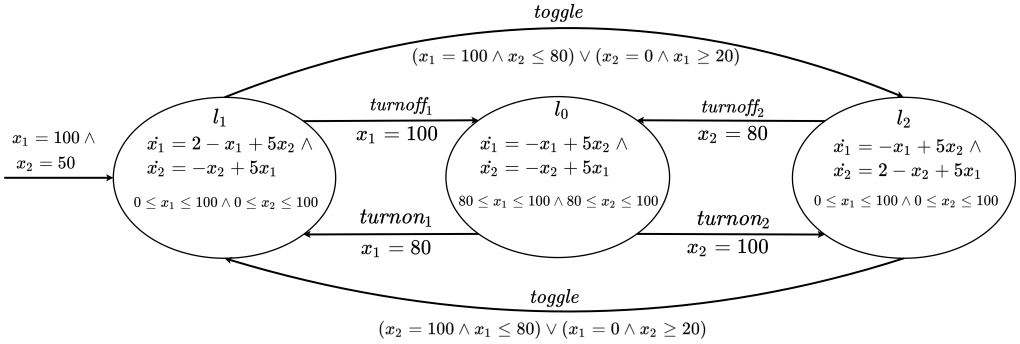



**Figure 3.1:** A hybrid automaton modelling a gas-burner.

The automaton has three locations $l_0, l_1$ and $l_2$ and two variables $x_1$ and $x_2$, the temperature in the two tanks ($\dot{x}_1$ and $\dot{x}_2$ denote the first derivative of $x_1$ and $x_2$, respectively). The gas-burner can be either switched off (in $l_0$) or turned on heating one of the two tanks (in $l_1$ or $l_2$). The dynamics in each location is given by a combination of the predicates that involve the derivatives of $x_1$ and $x_2$. On every edge of the automaton, we have omitted the condition $x_1' = x_1 \wedge x_2' = x_2$.

Basically, starting in location $l_1$, the burner heats up tank 1 until it reaches a temperature of 100 degrees. Since the temperature of tank 2 is below 80 degrees, the automaton takes edge *toggle* to location $l_2$. Note that edge $turnoff_1$ cannot be taken since the evolution $80 \leq x_1, x_2 \leq 100$ domain of the target location is not satisfied by $x_2$. In location $l_2$, the burner heats up tank 2 until it reaches 100 degrees. Since $x_1$ is still above 80 degrees, the automaton takes edge $turnoff_2$ to location $l_0$, where the burner is off. It

briefly remains here until $x_1$ falls to 80 degrees, at which point it takes edge $turnon_1$ to location $l_1$, where the burner heats up tank 1. The automaton converges towards a limit cycle of heating tank 1, heating tank 2, and briefly turning off the burner.

**Semantics** The behaviour of a hybrid automaton or how its state evolves over time is described by the semantics. In practice, at any time instant, the state of a hybrid system is given by a location $l$ and values for all variables of $X$. The state can change in two ways: (i) by a discrete and instantaneous transition that changes both the location $l$ and the values of the variables according to the transition relation. (ii) By a time delay that changes only the values of the variables $X$ according to the $\text{Flow}(l)$ of the current location $l$. The automaton may stay at a location $l$ only if the location invariant $\text{Inv}(l)$ is true, that is, some discrete transition must be taken before the invariant becomes false.

As regards the formal definition of the *semantics* of hybrid automata, for each hybrid automaton there is an associated transition system which formally defines the states and the transitions of the corresponding hybrid automaton. Formally, given the hybrid automaton $H = (\text{Loc}, \text{Lab}, \text{Edg}, X, \text{Init}, \text{Inv}, \text{Flow})$, its semantics is given in terms of the transition system $Ts(H) = \langle S, S_f, S_0, \Sigma, \rightarrow \rangle$ where $S = \{(l, v) \in \text{Loc} \times \mathbb{R}^X \mid v \in [\![\text{Inv}(l)]\!]\}$, $S_0 = \{(l, v) \in \text{Loc} \times \mathbb{R}^X \mid v \in [\![\text{Init}(l)]\!]\}$ and $S_f = \{(l, v) \in \text{Loc} \times \mathbb{R}^X \mid v \in [\![\text{Final}(l)]\!]\}$. The set actions associated to $Ts(H)$ is $\Sigma = \text{Lab} \cup \{(\mathsf{time}, \mathsf{r})\}$, where $\mathsf{time} \notin \text{Lab}$ and $r \in \mathbb{R}^{\geq 0}$. Finally, the transition relation $\rightarrow$ contains all the tuples $((l, v), \sigma, (l', v'))$ such that either:

- *discrete transition*. there exists $e = (l, \sigma, l') \in \text{Edge}$ such that $(v, v') \in [\![\text{Jump}(e)]\!]$, or
- *continuous transition*. $l = l'$ and $\sigma = (\mathsf{time}, r)$ for $r \in \mathbb{R}^{\geq 0}$ such that there exists a continuously differentiable function $\xi : [0, r] \rightarrow \mathbb{R}^X$ where $\xi(0) = v$ and $\xi(r) = v'$ and $(\xi(t), \dot{\xi}(t)) \in [\![\text{Flow}(l)]\!]$ for all $t \in [0, r]$ and $\xi(t) \in [\![\text{Inv}(l)]\!]$ for all $t \in [0, r]$. In particular, $\xi$ is called a *trajectory* from $v$ to $v'$. Usually $\text{Flow}(l)$ is a differential equation, in which case $\xi$ is a solution of that differential equation.

**Reachability and the safety problem** A state $q = (v, l) \in S$ is *reachable* if there exists a finite path $q_0\sigma_0 \ldots \sigma_{n-1}q_n$ where $q_0 \in S_0$ and $q_n = q$ and $q_i\sigma_i q_{i+1} \in \rightarrow$ for all $i \in n$. The set of *reachable* states of $Ts$ is $\text{Reach}(Ts)$. Finally, when $S_f$ specifies the unsafe states of the automaton, the transition system $Ts$ is said to be *safe* if $\text{Reach}(Ts) \cap S_f \neq \emptyset$.

**Definition 2** (Safety problem)**.** *Given a a hybrid automaton $H$, the* safety *(verification)* problem *for hybrid automata automata asks whether $Ts(H)$ is safe.*

Despite the somewhat simple formulation of the safety problem, many verification problems for hybrid systems reduce to the safety problem for hybrid automata. On the other hand, the safety problem is decidable only for restricted classes of hybrid automata. The main classes for which safety verification is decidable are timed automata, initialized rectangular automata, and o-minimal hybrid automata [77]. Here we focus only on initialized rectangular automata.

**Initialized rectangular automata** In order to present initialized hybrid automata we need to define *rectangular predicates*. A rectangular predicate over $X$ is an expression of the form a $a \prec x \prec b$, for $x \in X$, $\prec \in \{\leq, <\}$ and $a \leq b$ define a non-empty (possibly unbounded) interval with $a, b \in \mathbb{Q} \cup \{-\infty, +\infty\}$.

Rectangular automata are a subclass of hybrid automata with the following three restrictions:

1. the flow constraint in each location $l$ is a conjunction of rectangular predicates over $\dot{X}$;
2. the initial, final, and evolution domain conditions are conjunctions of rectangular predicates over $X$;
3. the jump condition of every edge is a conjunction of rectangular predicates over $X'$ and expressions of the form $x' = x$ for $x \in X$.

A hybrid automaton is *initialized* whenever every flow condition if it is changed for a variable $x$ by a discrete transition $e$, then this variable is (non-deterministically) reinitialized to a new value in $\text{Update}_e^x$ that is independent of the previous value. Formally, for every edge $e = (l, \sigma, l')$ and for every variable $x$ such that $\{v(\dot{x}) \mid v \in [\![\text{Flow}(l)]\!]\} \neq \{v(\dot{x}) \mid v \in [\![\text{Flow}(l')]\!]\}$, it holds that the set $\text{Update}_e^x(v) = \{v'(x') \mid (v, v') \in [\![\text{Jump}(e)]\!]\}$ does not depend on the valuation $v$, i.e., $\text{Update}_e^x(v) \neq \text{Update}_e^x(v')$ for all $v, v'$.

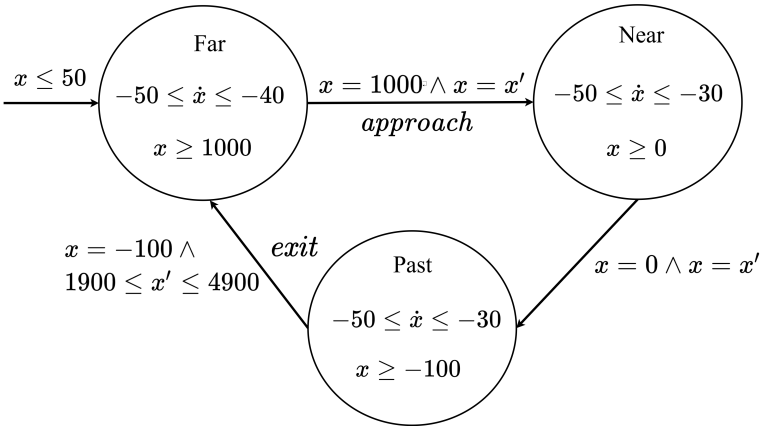



**Figure 3.2:** Initialized rectangular automaton of a train on a circular track with a gate.

As an example of a initialized rectangular automaton, in Figure 3.2 we show the automaton that models a train on a circular track with a gate [76]. In particular, the variable $x$ represents the distance of the train from the gate. Initially, the speed of the train is between 40 and 50 meters per second. At the distance of 1000 meters from the gate, the train issues an *approach* event and may slow down to 30 meters per second. At the distance of 100 meters past the gate, the train issues an *exit* event. The circular track is between 2000 and 5000 meters long.

### 3.1.2 Temporal logics

Temporal logics allow to express and check richer properties beyond safety. They are an extensions of propositional logics with operators that refer to the behaviour of systems over time which were originally developed by philosophers for investigating how time is used in natural languages.

Temporal logics are often classified according to whether time is assumed to have a *linear* or a *branching* structure and can specify a broad range of relevant properties such as: *functional correctness* (does the system do what is supposed to do), *safety* (something bad never happens), liveness (something good will eventually happen), fairness (does, under certain conditions, an event occur repeatedly?) and real-time properties (is the system acting in time?).

**Metric temporal logic** In this work we focus on *Metric Temporal Logic* (MTL) a particular kind of linear temporal logic (LTL) for hybrid systems where modalities are decorated with continuous timing constraints.

**Syntax** Let $P$ be a countable set of Boolean propositions and $I \subseteq [0, \infty)$ be an interval of reals with endpoints in $\mathbb{N} \cup \{\infty\}$, then the syntax of MTL is defined by the following grammar[1]:

$$\phi \quad ::= \quad \mathsf{true} \quad | \quad p \quad | \quad \neg\phi \quad | \quad \phi \vee \phi \quad | \quad \phi U_I \phi$$

**Semantics** As regards the semantics of MTL formulas, we show the the *pointwise semantics* where MTL formulas are interpreted over timed words [41]. The satisfaction of a formula is considered at a certain position of a model. MTL models are (finite or infinite) timed words $w = (a_1, t_n), \ldots, (a_n, t_n)$ where $a_i \in 2^P$ and $t_i \in \mathbb{R}$, for $1 \leq i \leq n$. Formally, given a (finite or infinite) timed word $w = (a_1, t_n), \ldots, (a_n, t_n)$ over the alphabet $2^P$ and an MTL formula $\phi$, the satisfaction relation $w, i \models \phi$ ($w$ satisfies $\phi$ at position $i$) is defined as follows:

$$\begin{array}{lll}
w, i \models \mathsf{true} & & \\
w, i \models p & \text{iff} & i <\mid w \mid \text{ and } a_i = p \\
w, i \models \neg\phi & \text{iff} & w, i \not\models \phi \\
w, i \models \phi_1 \vee \phi_2 & \text{iff} & w, i \models \phi_1 \text{ or } w, i \models \phi_2 \\
w, i \models \phi_1 U_I \phi_2 & \text{iff} & \text{there exists a } j \text{ such that } i < j <\mid w \mid, w, j \models \phi_2, t_j - t_i \in I, \\
 & & \text{and } w, k \models \phi_1 \text{ for all } k \text{ with } i < k < j
\end{array}$$

Of course, further Boolean connectives ($\wedge$, $\Longrightarrow$ and $\Longleftrightarrow$) can be defined following standard conventions. Furthermore, two temporal abbreviations are usually used: $\Diamond$ (eventually) and $\Box$ (globally). In particular, $\Diamond_I \phi \equiv \mathsf{true} U_I \phi$ states that there is

[1] Note that we show a fragment of MTL focusing on the future operators, the complete syntax containing also the past operators can be found in [41]

a location in the future in which $\phi$ holds and the timing constraint $I$ is satisfied, and $\Box_I \phi \equiv \neg\Diamond_I \neg\phi$ states that $\phi$ must hold in every location where the timing constraint $I$ is satisfied.

**Remark 1.** *When adopting the pointwise semantics it is common to think of atomic propositions in MTL as referring to events (corresponding to location changes) rather than to locations themselves.*

MTL formulas can express a number of useful properties, such as bounded-time *reachability*, e.g., $\Diamond_{\leq 4} start$, stating that within four time units the process must start. Alternatively, one can expresses bounded-time *recurrence*, e.g., $\Box\Diamond_{\leq 4} start$, stating that whatever the current state, within four time units the process will start. MTL can also be used to express bounded-time *response* properties, such as $\Box(start \implies \Diamond_{\leq 4} acquire)$.

### 3.1.3 Tools

Despite the restricted decidability of the safety problem, a number of formal verification tools for hybrid systems have been proposed, such as: `SpaceEx` [59], `PHAVer` [56] and `SpaceEx AGAR` [26], for linear/affine dynamics (i.e. continuous transitions expressed via linear/affine functions), and `HSolver` [132], `C2E2` [47] and `FLOW*` [36], for non-linear dynamics. Note that in order to to verify system where safety is undecidable, many of these tools adopt approximation techniques to obtain an estimation of the set of reachable states. Among these, the hybrid solver `PHAVer` addresses the *exact verification* of safety properties of hybrid systems with piecewise constant bounds on the derivatives, so-called *rectangular hybrid automata* [77]. *Affine dynamics* are handled by *on-the-fly over-approximation* and partitioning of the state space based on user-provided constraints and the dynamics of the system. Furthermore, performance comparisons have turned out in PHAVer's favor [106]. Indeed, to force termination and manage the complexity of the computations, methods to conservatively limit the number of bits and constraints are adopted.

## 3.2 Statistical model checking

*Statistical Model Checking* (SMC) [100] is a technique combining *model checking* [16] with the classical *Monte Carlo simulation* [74], aiming at providing support for quantitative analysis, as well as addressing the size barrier, to allow the analysis of large models. Unlike model checking, statistical model checking does not guarantee a 100% correct analysis, but it allows to bound the error of the analysis. In SMC, two statistical parameters $\alpha$ and $\epsilon$, lying in the real interval $[0, 1]$, must be specified by the user. The parameter $\alpha$ fixes the maximum probability of *false negatives*. On the other hand, $\epsilon$ fixes the probabilistic *uncertainty* of the analysis. Thus, an analysis in SMC returns a confidence interval $[\hat{p}-\epsilon,\ \hat{p}+\epsilon]$, in which $\hat{p}$ is the estimated probability that the model satisfies the checked property, where $\hat{p}$ satisfies the following constraint $Pr(\mid \hat{p} - p \mid \geq \epsilon) \leq \alpha$, where $p$ is the true probability of the model satisfying the checked property.

The number of runs that the simulator must perform to guarantee the level of required precision depends on both $\alpha$ and $\epsilon$, and it is computed using ad-hoc statistical techniques [100]. For instance, according to the Chernoff bound [120], $\alpha$ is related to the number of simulations $N$ by $\alpha = 2e^{-2N\epsilon^2}$, giving $N = (ln2 - ln\alpha)/(2\epsilon^2)$. Finally, the confidence interval (for given $\alpha$) may be derived with each new simulation measurement and the simulation generation is stopped when the confidence interval width is less than $2\epsilon$. A well known technique for that purpose is the Chow-Robbins sequential test [39].

## 3.3 Runtime enforcement

*Runtime enforcement* [135, 102, 51, 32] and *runtime verification* [53, 50, 101] are *dynamic verification* techniques that extract information from a *running* system and check *dynamically* if the observed behaviours satisfy properties of interest. In particular, runtime enforcement extends runtime verification and adopts an intrusive monitoring approach to ensure that the visible behaviour of the system under scrutiny (SuS) is always in agreement with some specification.

### 3.3.1 Runtime enforcement: a formal approach

The seminal work of Schneider [135] has introduced the notion of *security automata* to enforce security policies by terminating the system in case of a violation of the monitored property. Ligatti et al. have extended Schneider's work via edit automata [102], an enforcement mechanism capable of *suppressing* and *inserting* actions on behalf of the SuS. Furthermore, a number of other formalisations have been proposed, such as, *transducers* [10, 21], *shields* [84] or *enforcement-automata* [23, 51, 32].

In the formal approach, the monitors are often *synthesized* from a property to be enforced. The properties may be represented via automata-based formalisms [49, 51, 84, 127]. For instance, Falcone et al. [49, 51] proposed Streett automata to be translated into the respective enforcement automata. Pinisetty et al. [126] express the desired properties in terms of *Discrete Timed Automata* (DTA). On the other hand, in logic-based approaches, the properties are expressed as formulas of some logic. For instance, Martinelli and Matteucci [108] rely on the modal $\mu$-calculus (a reformulation of $\mu$HML). Cassar et al. use sHML, the safety subset of the branching time logic $\mu HML$ [32]. Though is not always the case that there is a clear separation between the property to be enforced and the enforcing monitor, for instance, in both of Ligatti's and Schneider's work properties are encoded in terms of the languages accepted by the enforcement model itself, i.e., as edit/security automata [32].

One important question in runtime enforcement is *enforceability*, i.e. , what kind of property can be enforced by a monitor at runtime? For instance, Schneider's security automata, can enforce only *safety* properties, as the automaton can only terminate the

system when the property is about to be violated. Edit automata are capable of enforcing instances of *safety* and *liveness* properties, along with other properties such as *infinite renewal* properties [103]. Monitors synthesized from Street automata [49, 51] can enforce most of the property classes defined within the *Safety-Progress hierarchy* [107].

The effectiveness of the enforcement depends on the achievement of the two following general principles [102]:

- *transparency*, *i.e.*, the enforcement must not prevent correct executions of the SuS;
- *soundness*, *i.e.*, incorrect executions of the SuS must be prevented.

Finally, enforcers may be distinguished in *uni-directional* or *bi-directional*. Uni-directional enforcers transform only the output behaviour of the SuS to ensure its correctness [135, 102, 49, 23]. Bi-directional enforcers correct the entire behaviour of the SuS, i.e, transforms both the outputs produced by the SuS and the inputs produced by the environment [126, 84, 32, 7].

### 3.3.2 Ligatti et al.'s edit automata

An *edit automaton* [102] is a finite or infinite state machine $(Q, q_0, \delta, \omega, \gamma)$ that is defined with respect to some system $S$. In particular, the system is defined as $S = (\mathcal{A}, \Sigma)$ where $\mathcal{A}$ is a set of actions and $\Sigma$ a set of possible executions, where an execution $\sigma \in \Sigma$ is a finite sequence of actions $a_1 \cdot a_2 \cdot \ldots \cdot a_n$, for $a_i \in \mathcal{A}$ and $1 \leq i \leq n$. The 5-tuple $(Q, q_0, \delta, \omega, \gamma)$ that specifies an edit automaton contains: the set of all possible states $Q$, the initial state $q_0$ and three partial functions:

- $\delta : \mathcal{A} \times Q \to Q$, specifies the transition function of the edit automaton;
- $\omega : \mathcal{A} \times Q \to \{-, +\}$, indicates whether or not the action in question is to be suppressed $(-)$ or emitted $(+)$;
- $\gamma : \mathcal{A} \times Q \to \overrightarrow{\mathcal{A}} \times Q$, specifies the insertion of a finite sequence of actions into the program's action sequence, where the first component in the returned pair $(\overrightarrow{\mathcal{A}})$ indicates the finite and non-empty sequence of actions to be inserted.

In order to maintain the determinacy, the partial functions $\delta$ and $\omega$ have the same domain while $\delta$ and $\gamma$ have disjoint domains. The operational semantics is defined according to the following rules:

$$\text{(Allow)}\ \frac{\sigma = a \cdot \sigma' \quad \delta(a, q) = q' \quad \omega(a, q) = +}{(\sigma, q) \xrightarrow{a} (\sigma', q')}$$

$$\text{(Suppress)}\ \frac{\sigma = a \cdot \sigma' \quad \delta(a, q) = q' \quad \omega(a, q) = -}{(\sigma, q) \to (\sigma', q')}$$

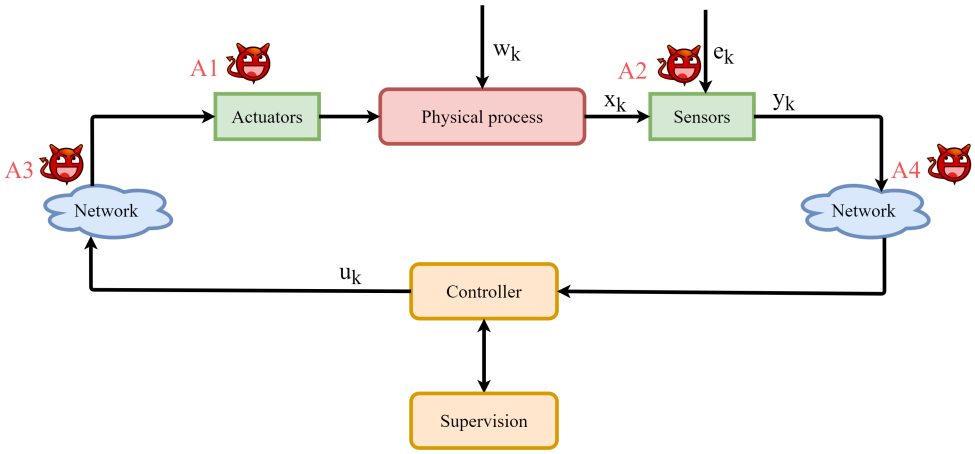
$$\text{(Insert)}\ \frac{\sigma = a \cdot \sigma' \quad \gamma(a, q) = \sigma'', q'}{(\sigma, q) \xrightarrow{\sigma''} (\sigma, q')}$$

Rule (Allow) is used for allowing actions emitted by the system under scrutiny. By Rule (Suppress) incorrect actions emitted by the system under scrutiny, are suppressed. Rule (Insert) is used to insert some finite and non-empty sequence of actions $\sigma''$ before an action $a$.

# PART I: An impact analysis for the security of control systems

**Threat model of part I** In the first part of the thesis we focus on *attacks targeting sensors and/or actuators* via either the corresponding physical device (A1 and A2) or the *communication network* used by the *device* (A3 and A4), i.e., *man-in-the-middle* attacks (see the figure below). Such attacks may manipulate the sensor measurements and/or the controller commands:

- attacks on sensors, i.e., reading and possibly replacing the genuine sensor measurements $y_k$ with fake ones $y_k^a$;
- attacks on actuators, i.e., reading, dropping and possibly replacing the genuine controller commands $u_k$ with malicious ones $u_k^a$.




Finally, recall that we assume that the attacker has already obtained access to the control system, and we do not consider the particular mechanisms of how vulnerabilities are exploited, and how the attack is hidden. Thus, we only consider the final objective of the attack, i.e., to maliciously affect the physical part.

# Chapter 4

# A Model Checking Approach

This chapter has the following structure. In Section 4.1 we first describe and then implement in HMODEST an engine system [94]. In Section 4.1.1 we put under stress the safety model checker prohver for a security analysis of the considered case study under three different cyber-physical attacks. Note that, as we describe in a quite detailed manner our implementation, we do not explain here the main syntactic constructs of the modelling language HMODEST. The interested reader can find an overview of such constructs in the Appendix.

## 4.1 A simple engine with a cooling system

We now describe the case study introduced in [94], which is called *Sys*.

*Sys* is an ICS in which the temperature of an engine is maintained within a certain range by means of a cooling system controlled by a controller. The system is also equipped with an IDS that does runtime safety verification. While *Sys* is a quite simple control system example, it is actually far from trivial and designed to describe a wide number of attacks.

Let's describe both the physical and the cyber component of *Sys*. The physical environment of *Sys* is constituted by:

- a *variable temp*, initialised to 0, for the current temperature of the engine, the *evolution equation* of the temperature is $temp_{k+1} = temp_k + cool_k + w_k$, where $w_k \in [-0.4, +0.4]$ denotes the *uncertainty* associated to *temp*; thus the variable *temp* is increased (respectively, is decreased) of one degree per time unit if the cooling system is inactive (respectively, active) up to a bounded uncertainty $w_k$;
- a *sensor* measuring the temperature of the engine, with a *measurement equation* $sens_k = temp_k + e_k$, where $e_k \in [-0.1, +0.1]$ denotes the *noise* associated to the sensor;
- an *actuator* to turn on/off the cooling system.

We remark here that, for simplicity, in the description of the case study we use a discrete-time model, although in its implementation we will adopt a continuous notion of time.

As regards the cyber component, *Sys* is provided with two parallel components: *Ctrl* and *IDS*. The former models the *controller* activity, consisting in reading the temperature of the engine and in governing the cooling system; whereas the latter models a simple *intrusion detection system* that attempts to detect and signal abnormal behaviours of the system.

*Ctrl* senses the temperature of the engine via the sensor (reads the sensor) at each time slot. When the *sensed temperature* is above 10 degrees, the controller activates the coolant via the actuator (sending a command to the actuator). The cooling activity is maintained for 5 consecutive time units. After that time, the controller synchronises with the *IDS* component, and then waits for *instructions* The *IDS* component checks whether the *sensed temperature* is still above 10. If this is the case, it sends an *alarm* of "high temperature", and then says to *Ctrl* to keep cooling for a further 5 time units; otherwise, if the temperature is not above 10, the *IDS* component requires *Ctrl* to stop the cooling activity.

In Figure 4.1, the left graphic collect 100 simulations of our engine in MATLAB, lasting 250 time units each, showing that the value of the state variable *temp* when the cooling system is turned on (*resp.*, off) lays in the interval $(9.9, 11.5]$ (*resp.*, $(2.9, 8.5]$); these bounds are represented by the dashed horizontal lines. The right graphic of the same figure shows three possible evolutions in time of the state variable *temp*: (i) the first one (in red), in which the temperature of the engine always grows as slow as possible and decreases as fast as possible; (ii) the second one (in blue), in which the temperature always grows as fast as possible and decreases as slow as possible; (iii) and a third one (in yellow), in which, depending whether the cooling is off or on, temperature grows or decreases of an arbitrary offset laying in the interval $[0.6, 1.4]$.

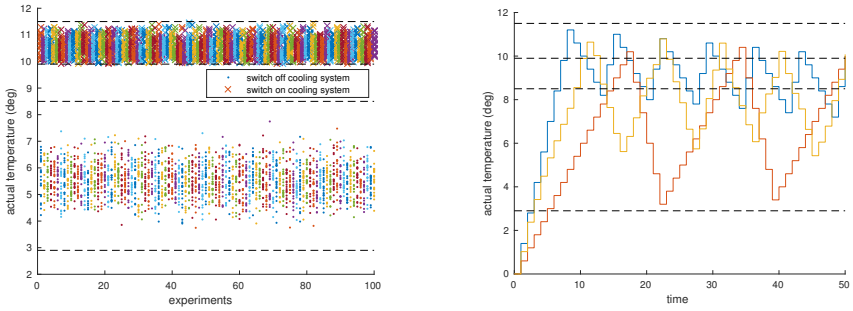



**Figure 4.1:** Simulations in MATLAB of *Sys*.

## Implementation in HModest

In this section, we provide our implementation in HMODEST of the case study presented in the previous section. The whole system is divided in three high level processes running in parallel (see Figure 4.2):

- *Plant()*, modelling the *physical aspects* of the system;
- *Logics()*, describing the *logical (or cyber) component*;
- *Network()*, representing the *network* connecting *Plant()* and *Logics()*.

The process *Plant()* consists of the parallel composition of four processes: *Engine()*, *Actuators()*, *Sensors()* and *Safety()* (see Figure 4.3). The former models the dynamics of the variable *temp* depending on the cooling activity. The temperature evolves in a continuous manner, and its rate is described by means of differential inclusions of the form $a \leq \dot{x} \leq b$ implemented via the construct *invariant*. The *on* action triggers the coolant and drives the process *Engine()* into a state *CoolOn()* in which the temperature decreases at a rate comprised in the range $[-\text{DT}-\text{UNCERT}, -\text{DT}+\text{UNCERT}]$. On the other hand, in the presence of an *off* action the engine moves into a *CoolOff()* state in which the coolant is turned off, so that the temperature increases at a rate ranging in $[\text{DT}-\text{UNCERT}, \text{DT}+\text{UNCERT}]$.

The second parallel component of *Plant()* is the process *Sensors()* that receives the requests to read the temperature, originating from the *Logics()*, and serves them according to the measurement equation seen in the previous section. This is modelled by updating the variable *sens* with an arbitrary real value laying in the interval $[temp - \text{NOISE}, temp + \text{NOISE}]$.

```
1  // global clock and global action declarations
2  clock global_clock;
3  action on, off;
4   ...
5  // global variable declarations
6  var sens = 0; der(sens) = 0;
7  bool safe = true;
8  bool is_deadlock = false;
9   ...
10 //process declarations
11 process Plant() {
12    var temp = 0;
13     ...
14    par { :: Engine() :: Sensors() :: Actuators() :: Safety() }
15 }
16 process Logics() {
17     ...
18    par { :: Ctrl() :: IDS() }
19 }
20 process Network() {
21     ...
22    par { :: Proxy_actuator() :: Proxy_sensor() }
23 }
24
25 // main
26 par { :: Plant() :: Logics() :: Network() }
```

**Figure 4.2:** Implementation in HMODEST of *Sys*.

The process Actuators() relays the commands of the controller *Ctrl()* to the *Engine()* to turn on/off the cooling system.

The last component of *Plant()* is the process *Safety()* which records the stress level of each physical variable (see Figure 4.3). Roughly, the stress level associated to the temperature is given by an integer number *stress* which keeps track of the consecutive time instants in which the controlled variables violate safety ranges. If this number exceeds 5 then the system is supposed to be in an unsafe state. Here, is worth mentioning that the variable *stress* could be implemented either as a bounded integer variable, which would increase the discrete complexity of the underlying hybrid automaton, or as a continuous variable with dynamics set to zero (*i.e.*, *der(stress)* = 0) that would increase the continuous complexity of the automaton. We have adopted the second option as it ensures better performances. The *Safety()* process sets the global Boolean variable *safe* to false only when the system reaches the maximum stress, *i.e.*, $stress = 5$, and reset it to true otherwise. Thus, this variable says when the ICS is currently in a state that is violating the *safety conditions*. Similarly, the global Boolean variable *is_deadlock* is set to true whenever the system invariant is violated; in that case the whole ICS stops.

```
1  const real DT = 1;
2  const real UNCERT = 0.4;   // uncertainty of variable temp
3  const real NOISE = 0.1;    // sensor noise
4  clock c;
5
6  process Engine() {
7      process CoolOn() {
8          invariant( der(temp) >= (-DT - UNCERT) && der(temp) <= (-DT + UNCERT) )
9          alt { :: on; CoolOn() :: off; CoolOff() }
10     }
11     process CoolOff() {
12         invariant( der(temp) >= (DT - UNCERT) && der(temp) <= (DT + UNCERT) )
13         alt { :: on; CoolOn() :: off; CoolOff() }
14     }
15     CoolOff()
16 }
17
18 process Sensors() {
19   do { // detect temperature and write it in variable sens
20        read_sensor {= sens = any(z, z >= temp - NOISE && z <= temp + NOISE), c = 0 =};
21        invariant(c <= 0) when(c >= 0) ack_sensor
22   }
23 }
24
25 process Actuators(){
26     do { :: cool_on_actuator {= c = 0 =}; invariant(c <= 0) when(c >= 0) on   // cool on
27          :: cool_off_actuator {= c = 0 =}; invariant(c <= 0) when(c >= 0) off   // cool off
28     }
29 }
30
31 process Safety() {
32     var stress = 0; der(stress) = 0;   // no continuous dynamics for stress
33     do { invariant(c <= 0) when(c >= 0)
34          alt { :: when(temp >= 0 && temp <= 20) // invariant is preserved
35                   alt { :: when(temp > 9.9 && stress <= 3) {= stress = stress+1 =}
36                         :: when(temp <= 9.9) {= stress = 0, safe = true =}
37                         :: when(temp > 9.9 && stress >= 4) {= stress = 5, safe = false =}
38                            // safety is violated
39                   }
40              :: when(temp > 20 || temp < 0) {= is_deadlock = true =}; stop // system deadlock
41          };
42          invariant(c <= 1) when(c >= 1) {= c = 0 =} // move to the next time unit
43     }
44 }
```

**Figure 4.3:** Plant() sub-processes.

The process *Logics()* consists of the parallel composition of two processes: *Ctrl()* and *IDS()* (see Figure 4.4). Concerning the behaviour of the former, senses the temperature by triggering a *read_sensor_ctrl* action to request a measurement and waits for an *ack_sensor_ctrl* action to read the measurement in the variable *sens*. Depending on the value of *sens* the controller decides whether to activate or not the cooling system. If $sens \leq 10$ the process sleeps for one time unit and then check the temperature again. If $sens > 10$ then the controller activates the coolant by emitting the *set_cool_on* action that will reach the *Engine()* (via the *Network()*'s proxy). Afterwards the control passes to the process *Check()* that verifies whether the current cooling activity is effective in dropping the temperature below 10. The process *Check()* maintains the cooling activity for 5 consecutive time units. After that, it synchronises with the process *IDS()* via the action *sync_ids*, and waits for instructions from *IDS()*: (i) *keep cooling* for other 5 time units and then check again, or (ii) *stop the cooling* activity and returns. These two instructions are represented by means of the actions *keep_cooling* and *stop_cooling*, respectively.

The second component of the process *Logics()* is the process *IDS()*. The *IDS()* process waits for the synchronisation action *sync_ids* from *Check()*. Then, it triggers the action *read_sensor_ids* to request a measurement and waits for the *ack_sensor_ids* action to read the measurement. If $sens \leq 10$ it signals to *Ctrl()* to stop cooling (via the action), otherwise, if $sens > 10$, it signals to keep cooling and fires an *alarm* by setting a global Boolean variable *alarm* to true (for verification reasons we immediately reset this variable to false).

The process *Network()* consists of the parallel composition of *Proxy_actuator()* and *Proxy_sensor()* (see Figure 4.5). The former provides the remote actuation. Basically, it forwards the actuators commands originating from the process Ctrl() to the process *Actuators()*. The process *Proxy_sensor()* waits for requests of measurement originating from processes *Ctrl()* or *IDS()* (we use different actions for each of them) and relay these requests to the process *Sensor()* that implements the measurement equation. When the temperature has been detected an ack signal is returned and propagated up to the requesting process.

**Verification.** We conduct our safety verification using a notebook with the following set-up: (i) 2.8 GHz Intel i7 7700 HQ, with 16 GB memory, and Linux Ubuntu 16.04 operating system; (ii) MODEST TOOLSET Build 3.0.23 (2018-01-19).

In order to assess the correct functioning of our implementation, we verify a number of properties of our system *Sys* by means of the safety model checker prohver. Here, it is important to recall that prohver relies on the hybrid solver `PHAVer` which computes an *overapproximation* of the reachable states to ensure termination and accelerate convergence [56]. As a consequence, the probability returned by the verification of a generic property $Pmax(\Diamond_T e_{prop})$ is an upper bound of the exact probability, and hence it is significant only when equal to zero (*i.e.*, when the property is not satisfied). However, as our

```
clock c;
process Ctrl() {
  process Check() {
    do{ invariant(c <= 0) when(c >= 0) tau;
        invariant(c <= 5) when(c >= 5) {= c = 0 =}; // keep cooling for 5 time units
        invariant(c <= 0) when(c >= 0) sync_ids; // activate IDS
        alt { // wait for instructions
            :: keep_cooling {= c = 0 =}  // keep cooling a further 5 time units
            :: stop_cooling {= c = 0 =};
               invariant(c <= 0) when(c >= 0) set_cool_off; // turn off the coolant
               invariant(c <= 1) when(c >= 1) {= c = 0 =}; // move to the next time slot
               invariant(c <= 0) when(c >= 0) break // returns the control to Ctrl()
        }
    }
  }
  // main Ctrl()
  do { invariant(c <= 0) when(c >= 0) read_sensor_ctrl;  // request temperature sensing
       ack_sensor_ctrl {= c = 0 =};
       invariant(c <= 0) when(c >= 0)
       alt { :: when(sens <= 10) tau {= c = 0 =};    // temperature is ok
              invariant(c <= 1) when(c >= 1) {= c = 0 =}  // move to the next time slot
           :: when(sens > 10) set_cool_on {= c = 0 =};  // turn on the cooling
              invariant(c <= 0) when(c >= 0) Check() // check whether temperature drops
       }
  }
}

process IDS() {
  do{ sync_ids {= c = 0 =};
      invariant(c <= 0) when(c >= 0) read_sensor_ids; // request temperature sensing
      ack_sensor_ids;
      invariant(c <= 0) when(c >= 0)
      alt { :: when(sens <= 10) stop_cooling // temperature is ok
          :: when(sens > 10) keep_cooling;  // temperature is not ok, keep cooling
             invariant(c <= 0) when(c >= 0) {= alarm = true =}; // fire the alarm
             invariant(c <= 0) when(c >= 0) {= alarm = false =}
      }
  }
}
```

**Figure 4.4:** Logics() sub-processes.

control system *Sys* presents a *linear dynamics* it is possible to compute the exact probability by launching our analyses with the NO_CHEAP_CONTAIN_RETURN_OTHERS flag (see [57]) which enables the exact computation of the reachable sets, with obvious implications on the time required to complete the analyses. As our case study does not present a probabilistic behaviour, the results of our analyses will always range in the set $\{0, 1\}$ (unsatisfied/satisfied) with a 100% accuracy.

Furthermore, as a formula $\Box e$ is satisfied if and only if $\Diamond \neg e$ is unsatisfied, we can use prohver to verify properties expressed in terms of formulae of the form $\Box_{[0,T]} e_{prop}$ or $\Diamond_{[0,T]} e_{prop}$. Actually, in our analyses we will always verify properties of the form $\Box_{[0,T]} e_{prop}$, relying on the quicker overapproximation when proving that the property is satisfied, and resorting to the slower exact computation when proving that the property is not satisfied.

Thus, we have formally proved that in all possible executions that are (at most) 100 time instants long the temperature of the system *Sys* oscillates in the real interval $[2.9, 11.5]$ (after a short initial transitory phase):

$$\Box_{[0,100]} (global\_clock \geq 5 \implies (temp \geq 2.9 \wedge temp \leq 11.5)) .$$

```
process Network() {
   clock c;
   process Proxy_actuator() {
      do { alt { :: set_cool_on {= c = 0 =};
                    invariant(c <= 0) when(c >= 0) cool_on_actuator
                 :: set_cool_off {= c = 0 =};
                    invariant(c <= 0) when(c >= 0) cool_off_actuator
           }
      }
   }
   process Proxy_sensor(){
      do { alt { :: read_sensor_ctrl {= c = 0 =};
                    invariant(c <= 0) when(c >= 0) read_sensor;
                    ack_sensor;
                    invariant(c <= 0) when(c >= 0) ack_sensor_ctrl
                 :: read_sensor_ids {= c = 0 =};
                    invariant(c <= 0) when(c >= 0) read_sensor;
                    ack_sensor;
                    invariant(c <= 0) when(c >= 0) ack_sensor_ids
           }
      }
   }

   par{ :: Proxy_actuator() :: Proxy_sensor() }
}
```

**Figure 4.5:** Network() process.

More generally, our implementation of *Sys* satisfies the following three properties:

- $\Box_{[0,100]}(\neg deadlock)$, saying that the system does not deadlock;
- $\Box_{[0,100]}(safe)$, saying that the system does not violate the safety conditions;
- $\Box_{[0,100]}(\neg alarm)$: saying that the IDS does not fire any alarm.

The verification of these three properties requires around 15 minutes each, thanks to the underlying overapproximation.

In the next section, we will verify our control system in the presence of three different cyber-physical attacks targeting either the sensor *sens* or the actuator *cool*. The reader can consult our models at `http://profs.scienze.univr.it/~merro/MODEST-FORTE/`.

### 4.1.1 Analyses and results

In this section we use the safety model checker prohver to perform a static security analysis of Sys. In particular, we implement three simple cyber-physical attacks targeting our system *Sys*:

- a *DoS* attack on the actuation mechanism that may push the system to violate the safety conditions and hence in the invariant conditions;
- a *DoS* attack on the sensor that may deadlock the system without being noticed by the IDS;
- an *integrity* attack on the sensor, again undetected by the *IDS*, that may drive the system into an unsafe state but only for a limited period of time.

These attacks are implemented by tampering with either the physical devices (actuators and/or sensors) or the communication network (*man-in-the-middle*). In order to

implement an attack on the sensor (*resp.*, actuator) we suppose the attacker is able to compromise the *Sensors()* (*resp.*, *Actuators()*) process. Whereas the attacks targeting the communication network compromise either the *Proxy_sensor()* or the *Proxy_actuator()* process, depending whether they are targeting the sensor or the actuator. In general, attacks on the communication network do not require a deep knowledge on the physical dynamics of the control system.

**Attack 1.** The first attack targets the actuator *cool* in a very simple manner. It operates exclusively in a specific time instant $m$, when it tries to drop the command to turn on the cooling system coming from the controller. Figure 4.6 shows the implementation of this man-in-the-middle attack compromising the *Proxy_actuator()* process.

```
1  process E_Proxy_actuator(){
2      clock c;
3          do{ alt{  ::  set_cool_on {= c = 0=};
4                        invariant(c <= 0) when(c >= 0)
5                        alt{  // drop the cool_on command in the time instant m
6                              ::  when(global_clock == m) tau
7                              // in the other time instants  forward correctly
8                              ::  when(global_clock < m || global_clock > m) cool_on_actuator
9                        }
10                    ::  set_cool_off  {= c = 0=};
11                       invariant(c <= 0) when(c >= 0) cool_off_actuator
12              }
13          }
14 }
```

**Figure 4.6:** DoS attack to the actuator.

We recall that the controller will turn on the cooling system only if it senses a temperature above 10 (as NOISE $= 0.1$, this means $temp > 9.9$). It is not difficult to see that this may happen only if $m > 7$ (in the time instant 7 the maximum temperature that may be reached by the engine is $7 \cdot (\mathrm{DT} + \mathrm{UNCERT}) = 7 \cdot (1 + 0.4) = 9.8$ degrees). Since the process *Ctrl()* never re-sends commands to the actuator, if the attacker is successful in dropping the command to turn on the cooling system in the time slot $m$ then the temperature will continue to rise, and after 2 time instants, in the time instant $m + 2$, the system will violate the safety conditions. This is noticed by the *IDS()* that will fire alarms every 5 time instants, until the system deadlocks because $temp > 20$.

We have verified the same properties stated in the previous section for the system *Sys* in isolation. None of those properties holds when the attack above operates in an instant $m > 7$. In particular, for $m > 7$ the system becomes unsafe in the time instant $m + 2$, and the *IDS()* detects the violation of the safety conditions with a delay of only 2 time instants. Summarising:

| Attack 1: tested properties | $m \leq 7$ | $m > 7$ |
|---|---|---|
| $\Box_{[0,100]}(\neg deadlock)$ | ✔ | ✖ |
| $\Box_{[0,100]}(safe)$ | ✔ | ✖ |
| $\Box_{[0,100]}(\neg alarm)$ | ✔ | ✖ |
| $\Box_{[0,m+1]}(safe)$ | ✔ | ✔ |
| $\Box_{[0,m+2]}(safe)$ | ✔ | ✖ |
| $\Box_{[0,m+3]}(\neg alarm)$ | ✔ | ✔ |
| $\Box_{[0,m+4]}(\neg alarm)$ | ✔ | ✖ |

As shown in this table, for $m \leq 7$ this is a *lethal attack* as it causes a deadlock of the system, however, it is not a *stealthy attack*. On the other hand, for $m > 7$ the attack is not *lethal* but it will bring the system into an unsafe state. Furthermore, an alarm will be fired, thus the attack is not *stealthy* in this case as well.

The properties above have been proved for all discrete time instants $m$, with $0 \leq m \leq 96$. The longest among these analyses required 20 minutes when overapproximating and at most 7 hours when doing exact verification.

**Attack 2.** The second attack compromises the sensor in order to provide fake measurements to the controller. The compromised sensor operates as follows: (i) in any time instant smaller than or equal to 1 the sensor works correctly, (ii) in any time instant greater than 1 the sensor returns the temperature sensed at time 1. Figure 4.7 provides an implementation of the compromised sensor.

```
1  process E_Sensors(){
2    clock c;
3    do{
4      alt{  :: when(global_clock <= 1) //normal behaviour
5               req_sensor {= sens = any(z, z >= temp−NOISE && z <= temp+NOISE), c = 0 =};
6               invariant(c <= 0) when(c >= 0) ack_sensor
7             :: when(global_clock > 1) //attack
8               req_sensor {= c = 0 =}; //the measurement remains unchanged
9               invariant(c <= 0) when(c >= 0) ack_sensor
10     }
11 }
```

**Figure 4.7:** DoS attack to the sensor.

In the presence of this attack, the process *Ctrl()* will always detect a temperature below 10 and never activate the cooling system or the IDS. The system under attack will move to an unsafe state until the system invariant will be violated and the system will deadlock. Indeed, in the worst case scenario, after $\lceil \frac{9.9}{\text{DT+UNCERT}} \rceil = \lceil \frac{9.9}{1.4} \rceil = 8$ time instants the value of *temp* will be above 9.9 degrees, and after further 4 time instants the system will violate the safety conditions. Furthermore, in the time instant $= \lceil \frac{20}{1.4} \rceil = 15$ the invariant may be broken and the system may deadlock because the state variable *temp* reaches 20.4 degrees.

The results of our security analysis are summarised in the following table:

| Attack 2: tested properties | |
|---|---|
| $\Box_{[0,100]}(\neg alarm)$ | ✔ |
| $\Box_{[0,100]}(safe)$ | ✖ |
| $\Box_{[0,100]}(\neg deadlock)$ | ✖ |
| $\Box_{[0,11]}(safe)$ | ✔ |
| $\Box_{[0,12]}(safe)$ | ✖ |
| $\Box_{[0,14]}(\neg deadlock)$ | ✔ |
| $\Box_{[0,15]}(\neg deadlock)$ | ✖ |

As shown in this table, this is a *lethal attack* as it causes a deadlock of the system. It is also a *stealthy attack* as it remains unnoticed until the end.

Concerning the performance of the analysis, the longest among these analyses required 35 minutes when overapproximating and at most 5 hours when doing exact verification. Notice that this attack does not require any specific knowledge of the sensor device (such as the measurement equation). Thus, the same goal could be obtained by means of a man-in-the-middle attack that compromises the *Proxy_sensor()* process.

**Attack 3.** Our last attack is a variant of the previous one as it provides the controller with a temperature decreased by an offset (in this case 2), for $n$ consecutive time instants. Unlike the previous attack, in case of encrypted communication, this attack cannot be mounted in the network as it requires the knowledge of the measurement equation. Figure 4.8 shows the implementation of a compromised sensor device acting as required. Basically, when $global\_clock <= n$ the compromised sensor returns a measurement affected by the offset; on the other hand, when $global\_clock > n$ the sensor works correctly and returns the authentic measurement.

```
process E_Sensors() {
    clock c;
    do { req_sensor {= c = 0 =};
         invariant(c <= 0) when(c >= 0)
         alt { :: when(global_clock <= n) //send corrupted measurement
                  {= sens = any(z, z >= (temp – 2 – NOISE) && z <= (temp – 2 + NOISE)),
                     c = 0 =};
               :: when(global_clock > n) //send authentic measurement
                  {= sens = any(z, z >= (temp – NOISE) && z <= (temp + NOISE)), c = 0 =}
         };
         invariant(c <= 0) when(c >= 0) ack_sensor
    }
}
```

**Figure 4.8:** Integrity attack to the sensor device.

The effects of this attack on the system depends on its duration $n$.

- For $n \leq 7$ the attack is harmless as the variable *temp* may not reach a (critical) temperature above 9.9; thus, all properties seen for the system in isolation remain valid when the system is under attack.

- For $n = 8$, the variable *temp* might reach a temperature above 9.9 and the attack would delay the activation of the cooling system of one time instant. As a consequence, the system might get into an unsafe state in the time instants 12 and 13, but no alarm will be fired (*stealthy attack*). This is proved by verifying the following properties:
  - $\Box_{[0,100]}((global\_clock < 12 \lor global\_clock > 14) \implies safe)$ ✔
  - $\Box_{[0,100]}((global\_clock \leq 12 \land global\_clock \geq 12) \implies safe)$ ✖
  - $\Box_{[0,100]}((global\_clock \leq 13 \land global\_clock \geq 13) \implies safe)$ ✖
  - $\Box_{[0,100]}(\neg alarm)$ ✔.
- For $n > 8$ the system may get into an unsafe state in a time instant between 12 and $n + 12$. The IDS will fire the alarm but it will definitely miss a number of violations of safety conditions as after the instant $n + 6$ it does not fire any alarm, although we prove there are unsafe states. This is a *temporary attack* as the system behaves correctly after the time instant $n + 12$. Summarising:
  - $\Box_{[0,100]}(\neg deadlock)$ ✔
  - $\Box_{[0,100]}((global\_clock < 12 \lor global\_clock > n + 12) \implies safe)$ ✔
  - $\Box_{[0,100]}((global\_clock \geq 12 \land global\_clock \leq n + 12) \implies safe)$ ✖
  - $\Box_{[0,100]}((global\_clock > n + 6 \land global\_clock \leq n + 12) \implies safe)$ ✖
  - $\Box_{[0,100]}((global\_clock < n+1 \lor global\_clock > n + 6) \implies \neg alarm)$ ✔
  - $\Box_{[0,100]}((global\_clock \geq n+1 \land global\_clock \leq n + 6) \implies \neg alarm)$ ✖.

The properties above have been proved for all discrete time instants $n$, with $0 \leq n \leq 85$. The longest among these analyses required 1 hour when overapproximating and at most 7 hours when doing exact verification.

## 4.2 Summary and discussion

**Summary** In this chapter we have implemented in HMODEST an ICS in which the temperature of an engine is maintained within a certain range by means of a cooling system controlled by a controller [94]. The system was also equipped with an IDS. Then we have proposed three significant *cyber-physical attacks* targeting the sensor or the actuator of the system: (i) a *DoS attack on the actuator* that operates as a man-in-the-middle on the connecting network; (ii) a *DoS attack on the sensor* that is achieved by compromising the sensor device; (iii) an *integrity attack on the sensor*, again by compromising the sensor device. Finally, we have tested the limits of the safety model checker prohver when doing static security analysis by means of three properties. These properties checked whether: (i) the system reached a deadlock state; (ii) the system reached an unsafe state; (iii) the IDS fired an alarm.

**Discussion** We have conducted a number formal security analyses in the context of CPSs via model-checking. Besides the safety model checker prohver, we have tried to verify the engine system [94] using other *model-checking tools*, such as PRISM [89], UPPAAL [19], and Real-Time Maude [121].

PRISM supports the verification of both CTL and LTL properties. This has allowed us to express proper formulae to verify violations the safety conditions, avoiding the implementation of the *Safety()* process. However, using integer variables to represent state variables with a fixed precision requires the introduction of extra transitions (to deal with nondeterministic errors) which significantly complicates the PRISM model.

In this respect, UPPAAL appears to be more effective than PRISM, as we have been able to concisely express the error occurring in integer state variables using the *select()* construct, in which the user can fix the granularity adopted to approximate a dense interval. This discrete representation provides an *under-approximation* of the system behaviour; although, a finer granularity translates into an exponential increase of the complexity of the system, with obvious consequences on the verification performance.

Then, we have tried to model our simple engine in Real-Time Maude, a completely different framework for real-time systems, based on *rewriting logic*. The language supports object-like inheritance features that are quite helpful to represent complex systems in a modular manner. We have used communication channels to implement our attacks on the physical devices. Furthermore, we have used rational variables for a more concise discrete representation of state variables. We have been able to verify LTL and T-CTL properties, although the verification process resulted to be quite slow due to a proliferation of rewriting rules when fixing a reasonable granularity to approximate dense intervals. As the verification logic is quite powerful, there is no need to implement an ad hoc process to check for safety.

At the end, as expected, a common aspect of the analyses conducted using these four model-checking tools (PRISM, UPPAAL, Real-Time Maude and prohver) was the performance limitations due to the well-known state explosion problem. Thus, when we have decided to step to the security analysis of a larger control systems, we have moved to statistical model checking, making a (small) compromise on the accuracy of the verification results. In this respect, the two main options were UPPAAL-SMC [44] and modes. In [96], we have tested UPPAAL-SMC for the security analysis of the simple engine verified in [91] using prohver, as shown in the previous section of this chapter. From these two experiences, we have realized that HMODEST provides us with a better and clearer representation of complex cyber-physical systems and cyber-physical attacks. Thus, in the following chapter we move to statistical model checking and we will show a security analysis of significantly larger ICS.

# Chapter 5

# A Statistical Model Checking Approach

In this chapter we test the effectiveness of statistical model checking for carrying out a static security analysis of an ICS in isolation and when exposed to cyber-physical attacks with different impacts. In particular, we use the *statistical model checker* modes of the MODEST TOOLSET. This chapter is structured as follows. In Sections 5.1 we first describe and then implement Johansson's tank system [81]. In Section 5.1.1, we do an SMC-based security analysis of three cyber-physical attacks.

## 5.1 A quadruple water tank system

In this section, we describe Johansson's tank system, a laboratory application (see Figure 5.1) where the cyber layer automatically controls, by means of two sensors and two pumps, the level of four interconnected water tanks. The water tank system maintains the water level of all four tanks within a fixed range. In order to achieve this goal, the system injects water into the four tanks by means of two pumps (Pump A and Pump B). The *controller* manages the injection of water by commanding the pumps.

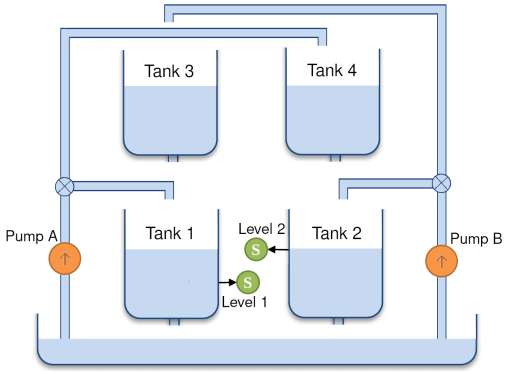



**Figure 5.1:** Johansson's quadruple-tank water system.

We enriched Johansson's original system with an *intrusion detection system* (IDS) to monitor malicious activities, and called again the resulting system *Sys*. The physical component, *i.e.*, the four interconnected water tanks, is described by means of four *nonlinear differential equations*. These equations relate the voltage applied to the pumps ($v_A, v_B$) to the four water levels ($h_1, h_2, h_3, h_4$). However, as modes cannot deal with nonlinear differential equations, we rely on a standard control-theoretic approach to approximate nonlinear differential equations with *linear difference equations* [45]. Thus, we adopt Johansson's discrete linear space model for his system [81], with a sampling time of $T_s = 10$ seconds. The model is obtained by instantiating the two linear difference equations described in the Introduction with the following matrices:

$$A = \begin{bmatrix} 0.8526 & 0 & 0.3144 & 0 \\ 0 & 0.8952 & 0 & 0.0888 \\ 0 & 0 & 0.6580 & 0 \\ 0 & 0 & 0 & 0.7165 \end{bmatrix}$$

$$B = \begin{bmatrix} 0.7695 & 0.0829 \\ 0.0149 & 0.5946 \\ 0 & 0.3910 \\ 0.2655 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

In these equations, $x_1$, $x_2$, $x_3$ and $x_4$ represent the state variables containing the values of the water levels governed by the commands $u_A$ and $u_B$. Notice that, due to the linearization process, the state variables $x_i$, for $i \in [1..4]$, and the commands $u_A$ and $u_B$ are related to the nonlinear physical variables and the voltage of the pumps in the following way: $x_i = h_i - h_i^0$, for $i \in [1..4]$ and $u_A = v_A - v_A^0$, $u_B = v_B - v_B^0$. Here, $h_i^0$, $v_A^0$ and $v_B^0$ are the operating points around which the nonlinear model has been linearized. Thus, whenever the value of $h_i$ is $h_i^0$, the value of $x_i$ is 0 (a similar reasoning can be done for $u_A$ and $u_B$). The evolution of each state variable $x_i$ is affected by some system uncertainty $w_i$, with $i \in [1..4]$, representing the *uncertainty* of the physical model. Furthermore, the variable $y_i$, for $i \in [1..2]$, provides the measurement of $x_i$ to the controller; each variable $y_i$ is affected by a measurement noise $e_i$. As concerns the initial condition of the variables, without loss of generality, we assume that the system is in a steady state, which means that $h_i = h_i^0$ (*i.e.*, $x_i = 0$ litres), for $i \in [1..4]$, in the linear model.

Let us now define the *cyber* component of *Sys*. It consists of two components running in parallel: the *Supervisor* and the *IDS*. The Supervisor monitors the evolution of the whole system; it consists of two parallel controllers:

- the first one governs the commands $u_A$, according to the value of $y_1$, in order to adjust the water flow given by Pump A (used to fill Tanks 1 and 4);
- the second one regulates the commands $u_B$, according to the value of $y_2$, in order to adjust the water flow given by Pump B (used to fill Tanks 2 and 3).

Note that controllers do not monitor directly Tank 3 and Tank 4. Indeed, Pump A regulates the inflow of Tank 1 and Tank 4, and if the water in Tank 1 lies in the interval $[-0.5, 0.5]$ then the water in Tank 4 will lie in $[-0.14, 0.14]$. In other words, $y_1$ provides

enough information to control Tanks 1 and 4 (a similar reasoning applies to $y_2$ and Tanks 2 and 3).

The IDS component simulates the physical evolution of the system by means of discrete linear difference equations, rising alarms when the values of the simulated variables differ, up to a proper threshold, from the real measurements. Furthermore, the IDS is capable of *mitigating* the effects of detected attacks by reconfiguring control commands to be sent to actuators via dedicated channels.

## An implementation in HModest

In this section, we provide our implementation in HMODEST. The whole implementation is divided in four main processes running in parallel (see Figure 5.2):

- *Global_Clock()*, modelling both the passage of time and process synchronization;
- *Plant()*, modelling both the *physical plant* and *physical devices*;
- *Logics()*, representing the *logical (or cyber) component*;
- *Network()*, connecting *Plant()* and *Logics()*.

Let us describe these processes in some detail.

```
1  // action, constant and variable declarations
2  action start_pump_a_actuator, read_level_1, sync_level_1_ids, ...;
3  const real NOISE_LVL1 = 0.007 , UNCERT_LVL1 = 0.015 , ...;
4  real level1, ... ,level4;
5  real sensed_level1, sensed_level2;
6  int stress_level1 = 0, ... , stress_level4 = 0;
7  bool safe_level1 = true, ... , safe_level4 = true;
8  bool deadlock_level1 = false, ... ,deadlock_level4 = false;
9  // process declarations
10 process Global_Clock() {
11     clock global_clock;
12     do {
13         evolution_step;
14         sync_process;
15         invariant(global_clock <= 1) when(global_clock >= 1) {= global_clock = 0 =}
16     }
17 }
18 process Plant() {
19     ... // local declarations
20     par { :: Physical_Process() :: Sensors() :: Actuators() :: Safety() }
21 }
22 process Logics() {
23     ... // local declarations
24     par { :: Supervisor() :: IDS() }
25 }
26 process Network() {
27     ... //proxies between Logics() and Plant()
28 }
29 // main
30 par { :: Global_Clock() :: Plant() :: Logics() :: Network() }
```

**Figure 5.2:** Implementation in HMODEST of *Sys*.

### The Global_Clock process

This process says to the other components when it is possible to perform an `evolution_step` action, modelling the evolution of physical variables. Then, a `sync_process` action is performed to synchronize all components of the system. At the end of each control cycle the global clock is reset.

### The Plant process

This process consists of the parallel composition of four sub-processes: *Physical_Process()*, *Sensors()*, *Actuators()* and *Safety()*.

The *Physical_Process()* models the physical evolution of the water tank system. This process waits for the `evolution_step` action to take place. When this action is triggered by *Global_Clock()* the process proceeds with the evolution of the variables $x_1$, $x_2$, $x_3$ and $x_4$ which are represented in terms of the HMODEST variables `level1`, `level2`, `level3` and `level4`, respectively. At the end, the process updates the value of the variables at the next time instant, according to the discrete time model given in Section 7.2. In order to model *uncertainty* of physical variables, we use random extractions of numbers, uniformly distributed, within a specific range.

The process *Sensors()*, given in Figure 5.3, implements the two sensors: the water level of Tank 1, via the variable `sensed_level1`, and the water level of Tank 2, via the variable `sensed_level2`. This process waits for requests coming from the controller to measure the levels. Again, the *measurement noise* is modelled by a random number extraction, uniformly distributed, within a specific range. Upon completion, the process sends an ack to the controller.

*Actuators()* executes the commands sent by the controller to the actuators (see Figure 5.3). In our model, this process sets the value of the control signals involved in the command just before the `evolution_step` action is performed by *Physical_Process()*. For instance, if the controller sends the command to turn off Pump A, then the actuator sets the control signal `PUMP_A_FLOW` to the associated correct value.

```
1  process Sensors() {
2      do {
3          :: read_level1 {= sensed_level1 = level1 + Uniform(−NOISE_LVL1,NOISE_LVL1) =};
4             ack_read_level1
5          :: read_level2 {= sensed_level2 = level2 + Uniform(−NOISE_LVL2,NOISE_LVL2) =};
6             ack_read_level2
7      }
8  }
9  process Actuators() {
10     do {
11         :: start_pump_a_actuator {= PUMP_A_FLOW = 0.2 =}
12         :: stop_pump_a_actuator {= PUMP_A_FLOW = −0.2 =}
13         :: start_pump_b_actuator {= PUMP_B_FLOW = 0.4 =}
14         :: stop_pump_b_actuator {= PUMP_B_FLOW = −0.4 =}
15     }
16 }
```

**Figure 5.3:** Sensors() and Actuators() processes.

The last component of *Plant()* is the process *Safety()* which records the stress level of each physical variable (see Figure 5.4). Roughly, the stress level associated to a physical variable is given by an integer number which keeps track of the consecutive time instants in which the controlled variables violate safety ranges. If this number exceeds a given threshold then the system is supposed to be in an unsafe state. The process consists of four parts. In Figure 5.4, we provide only the part concerning the water level of Tank 1, the other parts are similar. This process sets a global boolean variable `safe_level1` to false only when the system reaches the maximum stress, *i.e.*, `stress_level1` $= 4$, and resets it to true otherwise. Thus, this variable says when the system is currently in a state violating the *safety condition*. Similarly, the global boolean variable `deadlock_level1` is set to true whenever the system's invariant is violated (in that case, the whole stops).

```
process Safety() {
  do{
      alt {
        :: when((level1< −0.5 || level1> 0.5) && stress_level1 <= 2) {= stress_level1++ =}
        :: when(level1 >= −0.5 && level1<= 0.5) {= stress_level1 = 0, safe_level1 = true =}
        :: when((level1 < −0.5 || level1 > 0.5) && stress_level1 >= 3)
           {= stress_level1 = 4 , safe_level1 = false =}
        :: when(level1 < −1 || level1 > 1) {= deadlock_level1 = true =};
           stop
     };
        ... // similar for other variables
       evolution_step ;
       sync_process
  }
}
```

**Figure 5.4:** Safety() process.

### The Logics process

The *Logics()* process consists of two parallel sub-processes: *Supervisor()* and *IDS()*. *Supervisor()* contains two parallel controllers: *Controller_Level1()*, for the water level of Tank 1, and *Controller_Level2()*, for the water level of Tank 2. The *IDS()* consists of four parallel sub-processes: *Simulated_Plant()*, *Simulated_Actuators()*, *Reconfiguration_level1()*, *Reconfiguration_level2()*.

We will explain these processes by focusing on the control cycle of the variable `level1`, the water level of Tank 1 (see Figure 5.5). The machinery to manage the other three variables is similar. The goal of the controller is to maintain the water level of Tank 1 within the range $[-0.5, 0.5]$ (this interval has been empirically derived). Thus, the controller requests a measurement of the level to the associated sensor device. When the measurement is ready the sensor sends it to the controller. Both transmissions pass through the network and are acknowledged. The controller acts according to the received measurement: (i) if the level is below $-0.5$ then it commands to turn on Pump A, and informs the IDS about its decision; (ii) if the level is above 0.5 then it requests to turn off Pump A, and again informs the IDS about its decision; (iii) if the level lies within $[-0.5, 0.5]$ then no action is required. After that, a synchronization with the IDS is required to check for anomalies in the behaviour of Tank 1. At the end of the

```
process Controller_Level1() {
    do {
        read_level1_ctrl ; ack_read_level1_ctrl;
        alt {
            :: when(sensed_level1 < −0.5 ) start_pump_a_ctrl; ack_start_pump_a_ctrl; start_pump_a_ctrl2ids
            :: when(sensed_level1 > 0.5)  stop_pump_a_ctrl; ack_stop_pump_a_ctrl; stop_pump_a_ctrl2ids
            :: when(sensed_level1 >= −0.5 && sensed_level1 <= 0.5) tau // do nothing
        };
        sync_level1_ids; ack_level1_ids; evolution_step; sync_process
    }
}
```

**Figure 5.5:** Controller of the water level of Tank 1.

cycle, the physical variables are free to evolve (via the `evolution_step` action) and the controller will synchronize with the rest of the system (via the `sync_process` action).

As regards the *IDS()* process, the *Simulated_Plant()* sub-process (Figure 5.6) simulates the physical behavior of the water tanks and provides the IDS with the simulated state of *Physical_Process()*, *i.e.*, the simulated values of the water levels. The simulated physical variables are not affected by the system uncertainty, as they will be used to detect anomalies in the real physical variables.

The *Simulated_Actuators()* process of *IDS()* (Figure 5.6) actuates the *Simulated_Plant()* and it is designed to simulate the physical actuation. The simulated actuation must be updated/reconfigured whenever either the controllers update or the *IDS()* reconfigures the actuator commands. For instance, if the controller wants to change the value of `PUMP_A_FLOW` then it must synchronize with *Simulated_Actuators()* to change the value of `PUMP_A_FLOW_SIM`. The action `start_pump_a_ctrl2ids` is triggered whenever the controller of Tank 1 wants to start Pump A. Moreover, *Simulated_Actuators()* can also synchronize with *Reconfiguration_level1()* and *Reconfiguration_level2()* to replicate their commands. These commands are sent whenever a potentially unsafe behavior is detected.

```
process Simulated_Plant() {
    do {
        evolution_step {= level1_sim = level1_next_sim,  ...
        level1_next_sim = 0.8526*level1_sim + 0.3144*level3_sim
        + 0.7695*PUMP_A_FLOW_SIM + 0.0829*PUMP_B_FLOW_SIM, ... =};
        sync_process
    }
}
process Simulated_Actuators() {
    do {
        :: start_pump_a_ctrl2sim {= PUMP_A_FLOW_SIM = 0.2 =}
        ...
        :: start_pump_b_reconf2sim {= PUMP_B_FLOW_SIM = 0.2 =}
        ...
    }
}
```

**Figure 5.6:** IDS plant estimator.

Let us now describe the *Reconfiguration_level1()* process of the IDS (Figure 5.7). This process has two main activities: (i) monitoring Tank 1 for malicious activities that may

bring the water level out of the safety region, and (ii) reconfiguring controller commands when it is necessary to restore the safety of the tank. Thus, the process synchronizes with *Controller_Level1()* and then it acts depending on the distance between the current measurement of `level1` and its current simulated value. If that distance is greater than the threshold 0.2 then something wrong is supposed to be happening in the tank, and an alarm is risen. In this case, the safety of Tank 1 fully relies on the simulated values of its level. Thus, if the simulated `level1` is out of the safety interval $[-0.5, 0.5]$ then the reconfiguration process does two actions: (i) uses a dedicated channel shared with the actuators to reconfigure the controller commands, and (ii) reconfigures the simulated actuation as well, to be consistent with the real ones. Finally, the reconfiguration process synchronizes with the process *Controller_Level1()*. The implementation of *Reconfiguration_level2()* associated to the variable `level2` is similar.

```
process Reconfiguration_level1() {
    do {
        :: sync_level1_ids;
           alt {
               :: when(abs(level1_sim − sensed_level1) < 0.2) tau // do nothing
               :: when(abs(level1_sim − sensed_level1) >= 0.2) {= alarm_level1 = true =};
                  {= alarm_level1 = false =};
                  alt {
                      :: when(level1_sim < −0.5 ) start_pump_a_ids; ack_start_pump_a_ids;
                         start_pump_a_reconf2sim
                      :: when(level1_sim > 0.5) stop_pump_a_ids; ack_stop_pump_a_ids;
                         stop_pump_a_reconf2sim
                      :: when(level1_sim >= −0.5 && level1_sim <= 0.5) tau // do nothing
                  }
           };
           ack_level1_ids
    }
}
```

**Figure 5.7:** IDS control reconfiguration of Tank 1.

### The Network process

The last component of our implementation models the network infrastructure (see Figure 5.8). The *Network()* process consists of a set of *proxies* connecting physical and logical components. In particular, three main kinds of messages travel along the network:

1. sensor measurements;
2. control commands;
3. reconfiguration of control commands.

Thus, for instance, controllers use the network to transmit measurement requests which are forwarded to the appropriate sensor devices. When the measurements are ready they are transmitted via the network to the controllers which requested them. The transmission of both control commands from controllers to actuators and reconfiguration commands from IDSs and actuators works in a similar manner.

```
process Network () {
    do {
        // proxy for level1 measurements
        :: read_level1_ctrl; read_level1; ack_read_level1; ack_read_level1_ctrl
        // proxy for start commands to pump_b required by the controller
        :: start_pump_b_ctrl; start_pump_b_actuator; ack_start_pump_b_ctrl
        // proxy for reconfiguration commands to pump_b required by the IDS
        :: start_pump_b_ids; start_pump_b_actuator; ack_start_pump_b_ids
        // remaining proxies
        ...
    }
}
```

**Figure 5.8:** Network() process.

## Verification

We conduct our safety verification using a laptop with the following set-up: (1) 2.8 GHz Intel i7 7700 HQ, with 16GB memory, and Linux Ubuntu 16.04 operating system; (2) MODEST TOOLSET Build 3.0.141 (2019-04-17).

In order to assess the correct functioning of our implementation, we have verified a number of properties by means of the statistical model checker modes in which the confidence intervals are determined according to the Chow-Robbins sequential test [39], and the statistical parameters for false negatives and probabilistic uncertainty are both set to 0.01, to ensure a 99% accuracy. Basically, we have used modes to verify properties expressed in terms of formulae of the form $\Diamond_{[0,t]} e$, where $t$ is a positive integer. Thus, we compute the probability that a property $e$ holds in at least one time instant of the discrete time interval $[0, t]$. In particular, for all possible executions, that are at most 1000 time units long, we estimate the probabilities that the system *Sys* reaches three different *undesired states*: deadlock, violation of safety conditions, and alarm state. More precisely, we verify the following three properties:

- $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_\texttt{i})$, for $i \in \{1,2,3,4\}$, testing whether the system violates the invariant conditions;
- $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_\texttt{i})$, for $i \in \{1,2,3,4\}$, testing whether the system violates the safety conditions;
- $\Diamond_{[0,1000]}(\texttt{alarm\_level}_\texttt{i})$, for $i \in \{1,2\}$, testing whether the two IDSs fire an alarm for the two monitored variables.

The verification of the three properties above required a few minutes each.

### 5.1.1 Analyses and results

In what follows we use the statistical model checker modes to do an impact analysis of the system under investigation when exposed to three different cyber-physical attacks. In particular, we implement three simple, yet effective, attacks targeting the physical devices of our system *Sys*:

1. an *integrity attack* on the sensor of the water level of Tank 1;

2. a *DoS attack* on Pump B, preventing its deactivation;
3. a *MITM attack* on the network, combining a *replay attack* on the measurements of the water level of Tank 2, with a *DoS attack* on the Pump B.

These attacks are implemented by tampering either with the physical devices (sensors and/or actuators) of the system or with the communication network connecting logics and physical devices. In particular, the first attack compromises the *Sensors()* component, whereas both the second and the third attacks affect the *Network()* component.

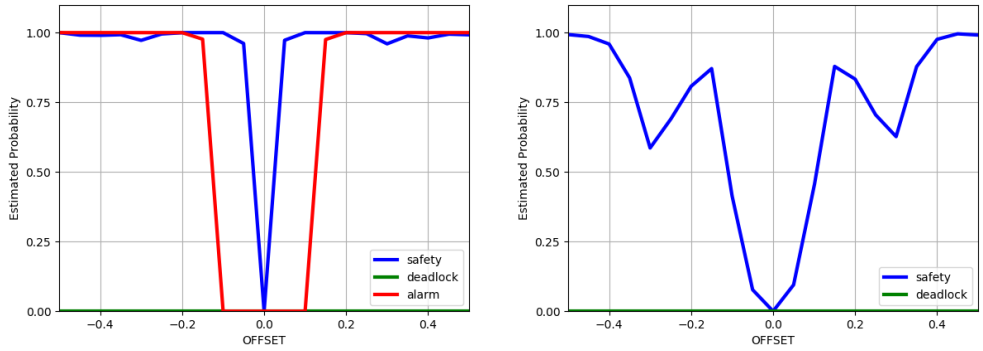
### Attack 1

We propose an attack altering the measurements of `level1` (contained in the variable `sensed_level1`) that the sensor device sends to the controller. The attack manipulates the measurements inside a *compromised Sensors()* device by adding an `OFFSET` (see Figure 5.9). Thus, during this attack the controller will base its decisions on wrong measurements of the level of Tank 1. As a consequence, it will delay the control actions deemed to keep the levels within the safety region. In our analysis, the offset added to the water level varies between $-0.5$ liters and 0.5 liters. Thus, for instance, if the attack introduces a negative offset then the controller may wrongly decide not to change the pump status, even if the actual level is 0.5, leading to an overflow of the corresponding tank. As Pump A regulates the water inflow in both Tank 1 and Tank 4, the attack will affect both tanks.

```
process Sensors_Integrity_Attack_Tank_1() {
  do {
     // compromised measurements of level1
     :: read_level1 {=sensed_level1=level1+OFFSET+Uniform(−NOISE_LVL1,NOISE_LVL1)=};
        ack_read_level1
     :: read_level2 {= sensed_level2 = level2  + Uniform(−NOISE_LVL2,NOISE_LVL2) =};
        ack_read_level2
    }
}
```

**Figure 5.9:** Integrity attack on the sensor level of Tank 1.

Figure 5.10 reports the impact of the attack on Tank 1 and Tank 4, respectively; the other two tanks are completely unaffected by the attack. As regards safety, Tank 1 will get into an unsafe state with probability 1 for most of the admitted offsets; while Tank 4 will reach an unsafe state with a significantly lower probability. Furthermore, the attack is not deadly, as the tanks never reach a deadlock state, *i.e.*, they do not overflow/underflow. As regards safety, for offsets in the range $[-0.1, 0.1]$ the attack may delay the control actions, leading to violations of the safety condition. This is a *stealthy attack* as the IDS will misinterpret the attack's forgery as process error and/or sensor noise. In fact, in this case, the offset is below the IDS threshold 0.2. On the contrary, for offsets outside the range $[-0.1, 0.1]$, the IDS detects the attack and fires an alarm with probability 1. Summarizing, the IDS leaves little margin for an attack




(a) Impact of attack 1 on Tank 1

(b) Impact of attack 1 on Tank 4

**Figure 5.10:** Red: $\Diamond_{[0,1000]}(\texttt{alarm\_level}_i)$ - Blue: $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_i)$ - Green: $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_i)$ - $i \in \{1,4\}$.

forging measurement data; furthermore, unnoticed attacks never drag the system into a deadlock state.

### Attack 2

Our second attack strikes Pump B by preventing the deactivation of the pump. We recall that this pump is used to fill both Tank 2 and Tank 3. As reported in Figure 5.11, the attack has been implemented by compromising the proxy of the network component associated to the actuator of Pump B. In particular, when a stop command arrives from the controller, the attack prevents the forwarding of the command to the actuator. The attack becomes operative at a random instant of time (variable `ATTACK_INIT`) chosen in the discrete time interval $[1, 500]$. The attack operates only within a finite time window (parameter `ATTACK_WND`). Furthermore, in order to fool the controller, an ack message is sent to it. In this way, both the controller and the IDS believe that the control command has been correctly dispatched.

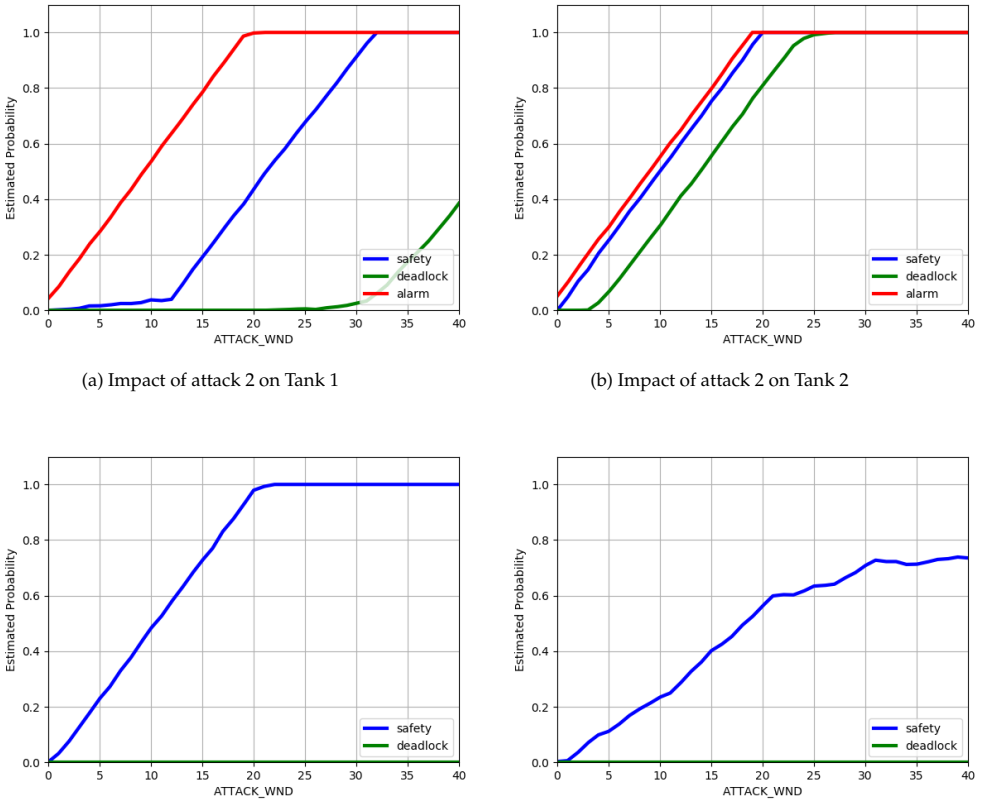
Figure 5.12 shows the impact of the attack on the four tanks, respectively. In our analysis the attack window may vary between 1 and 40 time instants. Let us examine first the tanks directly involved by Pump B: Tank 2 and Tank 3. As regards the safety of Tank 2, the probability to reach an unsafe state grows together with the size of the attack windows; this probability reaches 1 when the window reaches 20 time instants long. Furthermore, as Tank 2 may overflow, the attack may deadlock the whole system. Tank 3 violates its safety condition with the same pace as Tank 2. Finally, the IDS fires alarms with a probability slightly greater than safety violation, indicating that the detection works quite well.

Notice that the attack affects also Tank 1 and Tank 4 because more water is flowing from Tank 3. However, Tanks 1 and 4 may reach an unsafe state and/or deadlock with a significant smaller probability, when compared to Tank 2 and Tank 3. The IDS notices the anomaly on Tank 1 and it fires an alarm with a probability that is directly

```
process Network_DoS_Pump_B () {
    int ATTACK_INIT = DiscreteUniform(1,500);
    clock attack_clock;
    do {
        :: ... // unaffected proxies
        // compromised proxy
        :: stop_pump_b_ctrl;
            alt {// the proxy work correctly until ATTACK_INIT
                :: when(attack_clock <= ATTACK_INIT − 1) stop_pump_b_actuator
                :: when(attack_clock >= ATTACK_INIT)
                    alt {// in the attack window, drop the stop command
                        :: when(attack_clock <= ATTACK_INIT + ATTACK_WND) tau
                        // outside the attack window, forward the stop command correctly
                        :: when(attack_clock >= ATTACK_INIT + ATTACK_WND + 1)
                            stop_pump_b_actuator
                    }
        };
        ack_stop_pump_b_ctrl // send the ack to the controller
        :: ... // unaffected proxies
    }
}
```

**Figure 5.11:** DoS attack on Pump B to prevent its deactivation.

proportional to the size of the attack window. However, when monitoring the level of Tank 2, here we have a significant number of false positives as the threshold 0.2 of the




(a) Impact of attack 2 on Tank 1

(b) Impact of attack 2 on Tank 2

(c) Impact of attack 2 on Tank 3

(d) Impact of attack 2 on Tank 4

**Figure 5.12:** Red: $\Diamond_{[0,1000]}(\texttt{alarm\_level}_i)$ - Blue: $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_j)$ - Green: $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_j)$ - $i \in \{1,2\}, j \in \{1,2,3,4\}$.

IDS seems to be too tight for this kind of attack.

### Attack 3

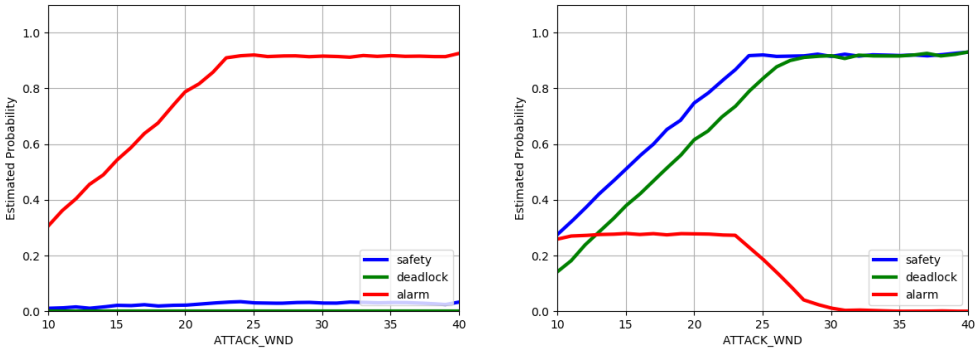
The third attack combines the *denial of service* on Pump B, as seen in Attack 2, together with an integrity attack on the measurements of the water level of Tank 2, by *replaying old measurements* that have been previously recorded. This is achieved by compromising two different proxies of the network: the proxy used to transmit stop commands to Pump B, and the proxy for requesting the measurements of `level2` (see Figure 5.13).

The attack consists of two phases: an *eavesdrop phase* and a *strike phase*. In the eavesdrop phase, the attacker records a sequence of measurements detected during an interval of time (`ATTACK_WND`). Then, in the attack window, the attacker replays the recorded measurements to the controller and simultaneously drops stop commands addressed to Pump B. The replay starts only when the measurements match the recorded measurements for two consecutive time units, up to an empirically determined tolerance 0.02.

```
process Network_DoS_Pump_B_Replay_Tank_2 () {
    int ATTACK_INIT = DiscreteUniform(1,500);
    clock attack_clock = 0;
    int i = 0, j = 0;
    real [] buffer = array(k,ATTACK_WND,0.0); // buffer of size ATTACK_WND
    do {// unaffected proxies
        :: ...
        // phase1: eavesdrop of measurements of level2
        :: read_level2_ctrl; read_level2; ack_read_level2;
        alt {// the proxy work correctly until ATTACK_INIT
            :: when(attack_clock < ATTACK_INIT) tau
            // start eavesdropping measurements at ATTACK_INIT
            :: when(attack_clock >= ATTACK_INIT)
            alt {// record measurements in the buffer
                :: when(i < ATTACK_WND) {= buffer[i] = sensed_level2, i++ =}
                // phase: strike−replay part
                :: when(i == ATTACK_WND)
                alt {// if the current measurement matches the recorded data up to 0.02
                    :: when(abs(sensed_level2 − buffer[j]) <= 0.02 && j < 2) {= j++ =}
                    // if the current measurement does not match the recorded data
                    :: when(abs(sensed_level2 − buffer[j]) > 0.02 && j < 2) {= j = 0 =}
                    //start replay attack
                    :: when(j >= 2 && j < ATTACK_WND) {= sensed_level2 = buffer[j], j++=}
                    // stop replay attack outside ATTACK_WND
                    :: when(j == ATTACK_WND) tau
                }
            }
        }; ack_read_level2_ctrl // send in any case the ack to the controller
        // phase2: strike, with drops of stop commands to Pump B
        :: stop_pump_b_ctrl;
        alt {// when the replay attack is going on
            :: when(j >= 2 && j < ATTACK_WND) tau // drop the stop command
            // when the replay attack stopped
            :: when(j < 2 || j >= ATTACK_WND) stop_pump_b_actuator
        }; ack_stop_pump_b_ctrl // send in any case the ack to the controller
    }
}
```

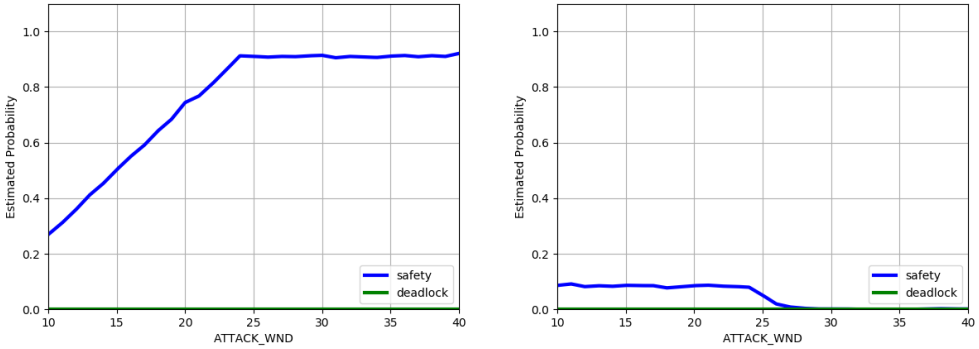
**Figure 5.13:** DoS on stop commands on Pump B together with a replay attack on measurements of Tank 2.

Figure 5.14 shows the impact of the attack on the four tanks, respectively. In our analysis the attack window may vary between 10 and 40 time instants. Let us examine the impact on the tanks directly involved by the attack: Tank 2 and Tank 3. As regards




(a) Impact of attack 3 on Tank 1

(b) Impact of attack 3 on Tank 2




(c) Impact of attack 3 on Tank 3

(b) Impact of attack 3 on Tank 4

**Figure 5.14:** Red: $\Diamond_{[0,1000]}(\texttt{alarm\_level}_i)$ - Blue: $\Diamond_{[0,1000]}(\neg\texttt{safe\_level}_j)$ - Green: $\Diamond_{[0,1000]}(\texttt{deadlock\_level}_j)$ - $i \in \{1,2\}, j \in \{1,2,3,4\}$.

the safety of Tank 2, an unsafe state is reached in all analyzed durations of the attack window; the probability of reaching an unsafe state in Tank 2 increases as the attack duration increases, with a direct proportional relation. Such probability reaches 0.95 when the time window is around 30 time instants long. This attack causes a violation of the invariant condition (deadlock) of Tank 2 with a high probability. As for the safety violations, this probability increases with the size of the attack window, although a bit more slowly. Tank 3 violates its safety condition with the same pace as Tank 2, but it never deadlocks. This is because the flow of water from Tank 3 to Tank 1 partially mitigates the effects of the attack. Thus, even if Pump B is not working correctly, the level of Tank 3 never violates the invariant condition. As regards the detection of the attack in Tank 2, the probability of firing an alarm decreases as the attack duration increases. Actually no alarms are thrown when the duration of the attack is longer than 30 time instants. This is because when the attack window is 30 time instants long the system deadlocks (*i.e.*, it stops completely) with a probability close to 1.

Finally, due to the physical interconnection of the tanks, the safety of Tank 1 and Tank 4 is slightly affected. However, these alterations are large enough to violate the

threshold $= 0.2$ of the IDS of Tank 1. In fact, although the replay attack on the measurements of the level of Tank 2 succeeds in cheating the corresponding IDS, this is not enough to implement a *stealthy attack*. A stealthy attack, *i.e.*, undetected by both IDSs, should tamper with the measurements of both Tank 1 and Tank 2.

## 5.2 Summary and discussion

**Summary** In this chapter we have implemented in HMODEST Johansson's tank system, a laboratory application where a controller regulates the level four interconnected water tanks, as shown in Figure 5.1. The water tank system maintains the water level of all four tanks within a safe range. This is achieved by means of tow pumps that inject water into the four tanks. Then we have evaluated the impact of three significant *cyber-physical attacks* in terms of deadlock, safety violations and IDS alarms, as presented in the Verification subsection. The attacks tampered with the sensor measurements or actuator commands via either the dedicated communication channels or the associated actuator/sensor devices. In particular, we have presented: (i) an *integrity* attack on a sensor of the system; (ii) a *DoS* attack on a pump, by tampering with its actuator; (iii) a combination of the previous attack with a *replay attack* on the measurements. Our analysis also provided us with insights on the performance of the proposed mitigating IDS. Finally, the analyses have been carried out on the *statistical model checker* modes of the MODEST TOOLSET.

**Discussion** The time required for the analyses was quite reasonable, in fact, on average each analysis required less than 15 minutes. The ICS under analyses contained four physical variables. Furthermore, recall that the statistical parameters for false negatives and probabilistic uncertainty were both set to 0.01, to ensure a 99% accuracy. On the other hand, as said in Chapter 4, the verifications required seven hours in some cases, there we have analysed an ICS with only one physical variable, some verifications required. Therefore, with a (small) compromise on the accuracy of the verification results, this confirms that modes and more generally statistical model checking can be successfully used to perform security analyses of non-trivial ICSs.

# Chapter 6

# End of Part I

In the first part of this thesis we have presented our investigations in applying static formal analysis techniques, i.e., model checking and statistical model checking, to perform and integrated security and safety analysis of ICSs when exposed to cyber-physical attacks targeting sensors and/or actuators via either the corresponding physical device or the communication network used by the device.

We have performed our analyses via the MODEST TOOLSET as it relies on a number of stat-of-the-art analysis backends. Furthermore, its modelling language, HMODEST, revealed to be very effective to specify realistic control systems in terms of four main parallel components: global clock, plant, logics and network. Such structured representation scales quite well and allowed us to represent cyber-physical attacks by focusing on specific compromised components (*e.g.*, sensors, actuators and network proxies).

In Chapter 4 we have implemented a simple but totally realistic and nuanced control system together with three cyber-physical attacks targeting the sensor or the actuator of the system. In particular, we have proposed: (i) a *DoS attack on the actuator* that operates as a man-in-the-middle on the connecting network; (ii) a *DoS attack on the sensor* that is achieved by compromising the sensor device; (iii) an *integrity attack on the sensor*, again by compromising the sensor device. Our implementation is quite clean and concise, although the current version of the language has still some problems in representing both instantaneous and delayed behaviours in an effective manner (we did not use the elegant *delay()* construct as each instance introduces a new clock, with heavy implications on the verification performance). Furthermore, in order to verify our safety and invariant conditions we have implemented a Safety() process that is not really part of our control system. From a designer point of view it would have been much more practical to use some kind of logic formula, such as: $\exists\Diamond(\Box_{[t,t+5]} temp > 9.9)$.

As said in the Introduction, the safety model checker within the MODEST TOOLSET relies on a modified version of the hybrid solver `PHAVer`, whose specification language is a slight variation of hybrid automata supporting compositional reasoning, where input and output variables are clearly distinguished [106]. Although, `PHAVer` would be a good candidate for the verification of small CPSs, we preferred to specify our case study

in the high-level language HMODEST, supporting: (i) differential inclusion to model linear CPSs with constant bounded derivatives; (ii) linear formulae to express nondeterministic assignments within a dense interval; (iii) compositional programming style inherited from process algebra (*e.g.*, parallel composition, nondeterministic choice, loops, *etc.*); (iv) shared actions to synchronise parallel components.

In Chapter 5 our goal was to test the statistical model checker modes as a tool for achieving a safety/security analysis of a non-trivial control system under attack. We have evaluated the physical impact of three carefully chosen *cyber-physical attacks* Our analysis also provided us with insights on the performance of the proposed mitigating IDS. The time required for the analyses, with a 99% accuracy, was quite reasonable (in average, less than 15 minutes for each analysis). This confirms that modes can be successfully used to perform security analyses of non-trivial ICSs. Again, in order to verify safety conditions we had to implement a *Safety()* process although it is not really part of the system under investigation. It would have been much more practical to use some formula as explained above.

## 6.1 Related work

In [145, 146], Vigo presents an attack scenario that addresses some of the peculiarities of a cyber-physical adversary, and discussed how this scenario relates to other attack models popular in the security protocol literature. Unlike us, this paper focuses on DoS attacks without taking into consideration timing aspects. Rocchetto and Tippenhaur [105] introduce a taxonomy of the diverse attacker models proposed for CPS security and outline requirements for generalised attacker models; in [133], they then propose an extended Dolev-Yao attacker model suitable for CPS security. In their approach, physical layer interactions are modelled as abstract interactions between logical components to support reasoning on the physical-layer security of CPSs. This is done by introducing additional orthogonal channels. Time is not represented. Nigam et al. [117] work around the notion of Timed Dolev-Yao Intruder Models for Cyber-Physical Security Protocols by bounding the number of intruders required for the automated verification of such protocols. Following a tradition in security protocol analysis, they provide an answer to the question: How many intruders are enough for verification and where should they be placed? They also extend the strand space model to CPS protocols by allowing for the symbolic representation of time, so that they can use Real-Time Maude [121] along with SMT support. Their notion of time is however different from ours, as they focus on the time a message needs to travel from an agent to another. The paper does not mention physical devices, such as sensors and/or actuators.

Pedroza *et al.* [125] proposed an UML-based environment to model critical embedded systems. The verification of safety properties relies on UPPAAL, whereas the verification of security properties (*e.g.*, confidentiality and integrity) relies on ProVerif [24]. Wardell *et al.* [149] proposed an approach for identifying security vulnerabilities of industrial control systems by modelling malicious attacks as PROMELA models. Kumar

*et al.* [87] introduced an attack-fault tree formalism to describe attack scenarios; they conduct formal analyses by using UPPAAL-SMC in order to obtain quantitative estimations on the impact of both system failures and security threats. Cheh *et al.* [35] used UPPAAL-SMC to do statistical model checking on a railway system to assess the safety of the system under attack. Like us, they tied safety analyses to security analyses and consider an attack that manipulates the communication messages exchanged between the signalling components of the railway system (this affects the speed of the trains and the routes that they take). More precisely, their attack is able to remove, insert, modify, or delay those network packets, much like a Dolev-Yao attacker. Huang *et al.* [78] analysed the impact of attacks compromising the safety of automotive systems. In particular, they considered attacks affecting the communications between vehicles: forgery of fake data, replay of old data or spoofing of vehicle IDs. Both safety and security properties are verified in UPPAAL-SMC. Taormina *et al.* [139] modelled the interaction between the physical and cyber layer of water distribution systems in epanetCPA, an open-source MATLAB toolbox allowing users to design custom cyber-physical attacks and simulate the hydraulic response of water networks. epanetCPA features a wide range of cyber-physical attacks: physical attacks to sensors and actuators, deception attacks, DoS on communication channels, replay attacks, and alteration of PLC and SCADA control statements. Although this tool can deal with several cyber-physical attacks, its analysis is based on single simulations.

In [115] the authors have proposed a formal approach to model and analyse the security and safety properties of ICSs. Both the ICS components and the attacker are modelled in the actor-based modelling language Rebeca, where the continuous physical dynamics are abstracted through discrete automata, for instance, different water levels (low, medium, and high) are modelled as discrete state variables. Thus, to model the increasing and decreasing of the water, the automaton performs a transition between these discrete levels delayed with a prefixed amount of time. Concerning security, the attacker can target: (i) the communication channels to inject malicious messages which may mislead the receiver and cause a system security failure; (ii) the ICS components: sensors, actuators and controllers to damage to damage them or alter the physical process; (iii) the combination of each. For such attacks, the security analysis verifies whether each tank will overflow or underflow. The modelling approach has been evaluated on the Secure Water Treatment (SWaT) System.

Lanotte *et al.* [93] provide a metric based on weak bisimulation metrics to estimate the impact of cyber-physical attacks targeting sensor devices of IoT systems formalised in a Timed Process Language. In [143] the authors propose and evaluation metric to analyse the trade-off between usability and security of cyber-physical attacks detection algorithms. In particular, the metric takes into account the impact of the worst attack that remains undetected and the average time between false alarms.

## 6.2 Future work

For the security analysis we have used LTL properties as specifications for the modelled system under attack. As future work we would like to analyse *time* properties regarding the responsiveness of the IDS to violations of safety conditions. Properties such as:

- there are integers $m, k$ such that the system may have an unsafe state at some instant $n > m$, and the IDS detects this violation with a delay of at least $k$ time instants ($k$ being a lower bound of the reaction time of the IDS);
- there is an instant $n$ where the IDS fires an alarm but neither an unsafe state nor a deadlock occurs between the instants $n - k$ and $n + k$; this would provide a tolerance of the occurrence of *false positive*.

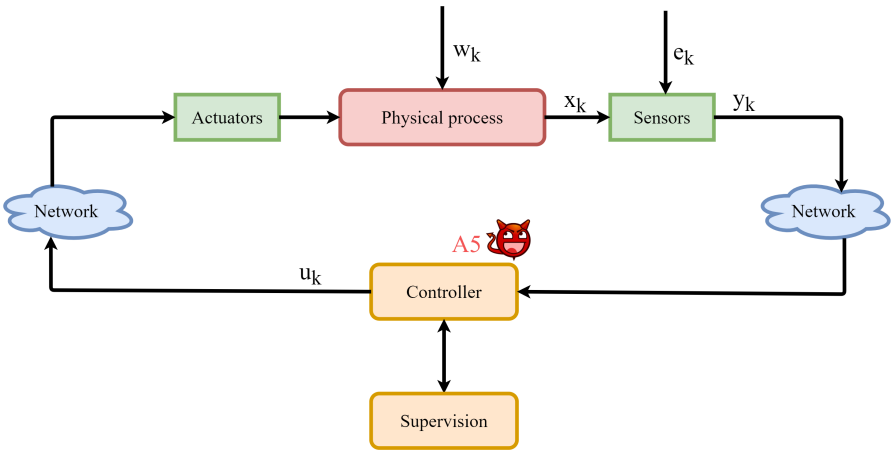
Although in general both model checking and statistical model checking techniques do not provide temporal properties of this form we could implement an ad hoc process to check these properties.

Another interesting future avenue is the security analysis through SMC of ICSs in which the physical plant shows a non-deterministic behaviour. Hybrid systems with nondeterministic physical behaviour can be modelled using Markov automata, and a number tools already support SMC for such automata [28].

Finally, the proposed method may appear to work only for known attacks against ICSs. However, due to the peculiar structure of ICSs, the research community has agreed that there are only a few types of attacks that can affect ICSs [61, 63, 141], the ones highlighted in Chapter 2. On the other hand, the parameters of cyber-physical attacks, such as timing, duration and offset, are not known. In this regard, we have proposed an efficient SMC-based method to analyse the impact of cyber-physical attacks with different parameters. Thus, an interesting future work would be to find a technique to efficiently analyse *colluding* cyber-physical attacks on complex ICS, i.e. attacks that compromise multiple sensors/actuators in order to achieve a goal. To do this we could reason compositionally, for example estimate the impact of attacks targetting a complex ICS in terms of impact on its subsystems.

# PART II: Runtime enforcement for control systems security

**Threat model of part II** In the second part of the thesis we focus on controllers compromised (A5) by *colluding malware* that may tamper with actuator commands, sensor readings, and inter-controller communications (see the figure below).




Again, we assume that the attacker has already obtained access to the control system, and we do not consider the particular mechanisms of how vulnerabilities are exploited, and how the attack is hidden. Instead, we consider the final objective of the attack, i.e., to maliciously affect the physical part.

# Chapter 7

# Runtime enforcement for control systems security: a formalisation

In this chapter we provide a formalisation of runtime enforcement tailored for the protection of control systems, i.e., the controllers. This chapter is structured as follows. Section 7.1 gives a formal language for monitored controllers. Section 7.2 defines the case study. Section 7.3 provides a language of regular properties to express controller behaviours; it also contains a taxonomy of properties expressible in the language. Section 7.4 contains the algorithm to synthesise monitors from regular properties, together with the main results.

## 7.1 The model

In this section, we introduce the *Timed Calculus of Monitored Controllers*, called TCMC, as an abstract formal language to express networks of controllers integrated with edit automata sitting on the network interface of each controller to monitor/correct their interactions with the rest of the system. Basically, TCMC extends Hennessy and Regan's *Timed Process Language* (TPL) [75] with monitoring edit automata. Like TPL time proceeds in *discrete time slots* separated by tick-actions.

Let us start with some preliminary notation. We use $s, s_k \in \mathsf{Sens}$ to name *sensor signals*; $a, a_k \in \mathsf{Act}$ to indicate *actuator commands*; $c, c_k \in \mathsf{Chn}$ for *channels*; $z_1, z_k$ for *generic names*.

### 7.1.1 A process calculus representation for PLCs

In our setting, controllers are nondeterministic sequential timed processes evolving through three main phases: *sensing* of sensor signals, *communication* with other controllers, and *actuation*. For convenience, we use five different syntactic categories to distinguish the five main states of a controller: $\mathbb{C}\mathtt{trl}$ for initial states, $\mathbb{S}\mathtt{leep}$ for sleeping states, $\mathbb{S}\mathtt{ens}$ for sensing states, $\mathbb{C}\mathtt{om}$ for communication states, and $\mathbb{A}\mathtt{ct}$ for actuation states. In its initial state, a controller is a recursive process *waiting* for signal

stabilisation in order to start the sensing phase:

$$\begin{array}{rcl} \mathbb{Ctrl} \ni P & ::= & X \\ \mathbb{Sleep} \ni W & ::= & \mathsf{tick}.W \;\mid\; S \end{array}$$

The main process describing a controller consists of some *recursive process* defined via equations of the form $X = \mathsf{tick}.W$, with $W \in \mathbb{Sleep}$; here, $X$ is a *process variable* that may occur (free) in $W$. For convenience, our controllers always have at least one initial timed action tick to ensure *time-guarded recursion*, thus avoiding undesired *zeno behaviours*: the number of untimed transitions between two timed ones is always bounded. Then, after a determined sleeping period, when sensor signals get stable, the sensing phase can start.

During the sensing phase, the controller waits for a *finite* number of admissible sensor signals. If none of those signals arrives in the current time slot then the controller will *timeout* moving to the following time slot (we adopt the TPL construct $\lfloor\cdot\rfloor\cdot$ for timeout). The syntax is the following:

$$\mathbb{Sens} \ni S \quad ::= \quad \lfloor \textstyle\sum_{i\in I} s_i.S_i \rfloor S \;\mid\; C$$

where $\sum_{i\in I} s_i.S_i$ denotes the standard construct for nondeterministic choice. Once the sensing phase is concluded, the controller starts its calculations that may depend on *communications* with other controllers governing different physical processes. Controllers communicate with each other for mainly two reasons: either to receive notice about the state of other physical sub-processes or to require an actuation on a physical process which is out of their control. As in TPL, we adopt a *channel-based handshake point-to-point* communication paradigm. Note that, in order to avoid starvation, communication is always under timeout. The syntax for the communication phase is:

$$\mathbb{Comm} \ni C \quad ::= \quad \lfloor \textstyle\sum_{i\in I} c_i.C_i \rfloor C \;\mid\; \lfloor \overline{c}.C \rfloor C \;\mid\; A$$

Once the communication phase is over, the controller moves on to the actuation phase. In the actuation phase a controller eventually transmits a *finite* sequence of commands to actuators, and then, it emits a *fictitious control signal* end to denote the end of the scan cycle. After that, the whole scan cycle can restart. Formally,

$$\mathbb{Act} \ni A \quad ::= \quad \overline{a}.A \;\mid\; \mathsf{end}.X$$

**Remark 2** (Scan cycle duration and maximum cycle limit)**.** *The scan cycle of a PLC must be completed within a specific time, called* maximum cycle limit*, which depends on the controlled physical process; if this time limit is violated the controller stops and throws an exception [137]. Thus, the signal* end *must occur well before the* maximum cycle limit *of the controller. We actually work under the assumption that our controllers successfully complete*

$$\text{(Sleep)}\ \frac{-}{\mathsf{tick}.W \xrightarrow{\mathsf{tick}} W} \qquad \text{(Rec)}\ \frac{X = \mathsf{tick}.W}{X \xrightarrow{\mathsf{tick}} W}$$

$$\text{(ReadS)}\ \frac{j \in I}{\lfloor \sum_{i\in I} s_i.S_i \rfloor S \xrightarrow{s_j} S_j} \qquad \text{(TimeoutS)}\ \frac{-}{\lfloor \sum_{i\in I} s_i.S_i \rfloor S \xrightarrow{\mathsf{tick}} S}$$

$$\text{(InC)}\ \frac{j \in I}{\lfloor \sum_{i\in I} c_i.C_i \rfloor C \xrightarrow{c_j} C_j} \qquad \text{(TimeoutInC)}\ \frac{-}{\lfloor \sum_{i\in I} c_i.C_i \rfloor C \xrightarrow{\mathsf{tick}} C}$$

$$\text{(OutC)}\ \frac{-}{\lfloor \overline{c}.C \rfloor C' \xrightarrow{\overline{c}} C} \qquad \text{(TimeoutOutC)}\ \frac{-}{\lfloor \overline{c}.C \rfloor C' \xrightarrow{\mathsf{tick}} C'}$$

$$\text{(WriteA)}\ \frac{-}{\overline{a}.A \xrightarrow{\overline{a}} A} \qquad \text{(End)}\ \frac{-}{\mathsf{end}.X \xrightarrow{\mathsf{end}} X}$$

**Table 7.1:** LTS for controllers.

*their scan cycle in less than half of the maximum cycle limit. The reasons for this assumption will be clarified in Remark 4.*

The operational semantics in Table 7.1 is along the lines of Hennessy and Regan's TPL [75].

**Remark 3** (Time determinism)**.** *The operational semantics of our timeout construct, inherited from TPL [75], ensures us a useful property when synthesising monitors:* time determinism. *Basically, this property says that the execution of each* tick*-action leads to at most one new state.*

In the following, we use the metavariables $\alpha$ and $\beta$ to range over the set of all observable actions: $\{s, \overline{a}, \overline{c}, c, \mathsf{tick}, \mathsf{end}\}$. These actions denote: sensor readings, actuator commands, channel transmissions, channel receptions, passage of time, and end of scan cycles, respectively.

### 7.1.2 An enforcement mechanism based on edit automata

The core of our enforcement relies on (timed) finite-state Ligatti et al.'s *edit automata* [102], *i.e.*, a particular class of automata specifically designed to allow/suppress/insert actions in a generic system in order to preserve its correct behaviour. The syntax follows:

$$\mathbb{E}\mathrm{dit} \ni \mathsf{E} \quad ::= \quad \mathsf{go} \ \mid \ \sum_{i\in I} \lambda_i.\mathsf{E}_i \ \mid \ \mathsf{X}$$

The special automaton go will admit any action of the monitored system. The edit automaton $\sum_{i\in I} \lambda_i.\mathsf{E}_i$ enforces an action $\lambda_i$, and then continues as $\mathsf{E}_i$, for any $i \in I$, with $I$ finite. Here, the symbol $\lambda$ ranges over: (i) $\alpha$ to *allow* the action $\alpha$, (ii) $^-\alpha$ to *suppress* the action $\alpha$, and (iii) $\beta \prec \alpha$ to *insert* the action $\beta$ before the action $\alpha$.

Finally, *recursive automata* $\mathsf{X}$ are defined via equations of the form $\mathsf{X} = \mathsf{E}$, where the automata variable $\mathsf{X}$ may occur (free) in $\mathsf{E}$. The operational semantics of our edit

automata is the following:

$$\text{(Go)}\ \frac{-}{\mathsf{go} \xrightarrow{\alpha} \mathsf{go}} \qquad \text{(Edit)}\ \frac{j \in I}{\sum_{i \in I} \lambda_i.\mathsf{E}_i \xrightarrow{\lambda_j} \mathsf{E}_j} \qquad \text{(recE)}\ \frac{\mathsf{X} = \mathsf{E} \quad \mathsf{E} \xrightarrow{\lambda} \mathsf{E}'}{\mathsf{X} \xrightarrow{\lambda} \mathsf{E}'}$$

Our *monitored controllers*, written $\mathsf{E} \bowtie J$, consist of a controller $J$, for $J \in \mathbb{C}\mathtt{trl} \cup \mathbb{S}\mathtt{leep} \cup \mathbb{S}\mathtt{ens} \cup \mathbb{C}\mathtt{omm} \cup \mathbb{A}\mathtt{ct}$, and an edit automaton $\mathsf{E}$ enforcing the behaviour of $J$, according to the following transition rules, presented in the style of [108]:

$$\text{(Allow)}\ \frac{\mathsf{E} \xrightarrow{\alpha} \mathsf{E}' \quad J \xrightarrow{\alpha} J'}{\mathsf{E} \bowtie J \xrightarrow{\alpha} \mathsf{E}' \bowtie J'} \qquad \text{(Suppress)}\ \frac{\mathsf{E} \xrightarrow{\bar{\alpha}} \mathsf{E}' \quad J \xrightarrow{\alpha} J'}{\mathsf{E} \bowtie J \xrightarrow{\tau} \mathsf{E}' \bowtie J'}$$

$$\text{(Insert)}\ \frac{\mathsf{E} \xrightarrow{\beta \prec \alpha} \mathsf{E}' \quad J \xrightarrow{\alpha} J' \quad \mathsf{E} \xrightarrow{\alpha} \!\!\!\!\!/}{\mathsf{E} \bowtie J \xrightarrow{\beta} \mathsf{E}' \bowtie J}$$

Rule (Allow) is used for allowing observable actions emitted by the controller under scrutiny. By an application of Rule (Suppress), incorrect actions emitted by (possibly corrupted) controllers are suppressed. Rule (Insert) is used to insert an action $\beta$ before an action $\alpha$ (of the controller) which is not allowed by the monitoring automata.

Thus, in a monitored controller $\mathsf{E} \bowtie J$, when $J$ complies with the property enforced by $\mathsf{E}$, the two components $\mathsf{E}$ and $J$ evolve in a tethered fashion (by applying rule (Allow)), moving through related correct states. On the other hand, if $J$ gets somehow corrupted (for instance, due to the presence of a malware) then $\mathsf{E}$ and $J$ will get misaligned reaching unrelated states. In this case, the remaining actions emitted by the controller will be suppressed by the monitor $\mathsf{E}$ until the controller $J$ reaches the end of the scan cycle, signalled by the emission of the $\mathsf{end}$-action[1]. After that, if $\mathsf{E}$ and $J$ are not aligned then $\mathsf{E}$ will command the execution of a safe trace, without any involvement of the controller, via one or more applications of the rule (Insert). Safe traces inserted in full autonomy by our enforcers always terminate with an $\mathsf{end}$. Thus, when both the controller and the monitor will be aligned at the end of the scan cycle, they will synchronise on the action $\mathsf{end}$, via an application of the rule (Allow), and from then on they will continue in a tethered fashion.

**Remark 4.** *Note that in case of insertion of a safe trace by the monitor, the assumption made in Remark 2 ensures us that scan cycles always end well before a violation of the maximum cycle limit.*

Obviously, we can easily generalise the concept of monitored controller to a *field communications network* of parallel monitored controllers, each one acting on different

[1] In general, malware that aims to take control of the plant has no interest in delaying the scan cycle and risking the violation of the maximum cycle limit whose consequence would be the immediate controller shutting down [137].

$$\text{(ParL)}\ \frac{N_1 \xrightarrow{\alpha} N_1'}{N_1 \parallel N_2 \xrightarrow{\alpha} N_1' \parallel N_2} \qquad \text{(ChnSync)}\ \frac{N_1 \xrightarrow{c} N_1' \quad N_2 \xrightarrow{\bar{c}} N_2'}{\begin{array}{c} N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2' \\ N_2 \parallel N_1 \xrightarrow{\tau} N_2' \parallel N_1' \end{array}}$$

$$\text{(ParR)}\ \frac{N_2 \xrightarrow{\alpha} N_2'}{N_1 \parallel N_2 \xrightarrow{\alpha} N_1 \parallel N_2'} \qquad \text{(TimeSync)}\ \frac{N_1 \xrightarrow{\mathsf{tick}} N_1' \quad N_2 \xrightarrow{\mathsf{tick}} N_2' \quad N_1 \parallel N_2 \xrightarrow{\tau} \not\rightarrow}{N_1 \parallel N_2 \xrightarrow{\mathsf{tick}} N_1' \parallel N_2'}$$

**Table 7.2:** LTS for field communications networks of monitored controllers.

actuators, and exchanging information via channels. These networks are formally defined via a straightforward grammar:

$$\mathbb{FN}\mathbb{e}\mathbb{t} \ni N \ ::= \ \mathsf{E} \ltimes J \ \mid \ N \parallel N$$

with the operational semantics defined in Table 7.2.

Notice that monitored controllers may interact with each other via channel communication (see Rule (ChnSync)). Moreover, via rule (TimeSync) they may evolve in time only when channel synchronisation may not occur (our controllers do not admit zeno behaviours). This ensures *maximal progress* [75], a desirable time property when modelling real-time systems: channel communications will never be postponed.

Having defined the possible actions $\beta$ of a monitored field network (we recall that $\beta$ may also range over $\tau$-actions, due to an application of rule (Suppress)), we can easily concatenate actions to define *execution traces*.

**Definition 3** (Execution traces)**.** *Given a finite execution trace $t = \beta_1 \dots \beta_k$, we write $N \xrightarrow{t} N'$ as an abbreviation for $N = N_0 \xrightarrow{\beta_1} N_1 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_{k-1}} N_{k-1} \xrightarrow{\beta_k} N_k = N'$.*

In the rest of the chapter we adopt the following notations.

**Notation 1.** *As usual, we write $\epsilon$ to denote the* empty trace*. Given a trace $t$ we write $\mid t \mid$ to denote the* length *of $t$,* i.e.*, the number of actions occurring in $t$. Given a trace $t$ we write $\hat{t}$ to denote the trace obtained by removing the $\tau$-actions. Given two traces $t'$ and $t''$, we write $t' \cdot t''$ for the trace resulting from the* concatenation *of $t'$ and $t''$. For $t = t' \cdot t''$ we say that $t'$ is a* prefix *of $t$ and $t''$ is a* suffix *of $t$.*

## 7.2 Use case: the SWaT system

In this section, we describe how to specify in TCMC a non-trivial network of PLCs to control (a simplified version of) the *Secure Water Treatment system* (SWaT) [109].

SWaT represents a scaled down version of a real-world industrial water treatment plant. The system consists of 6 stages, each of which deals with a different treatment, including: chemical dosing, filtration, dechlorination, and reverse osmosis. For simplicity, in our use case, depicted in Figure 7.1, we consider only three stages. In the first

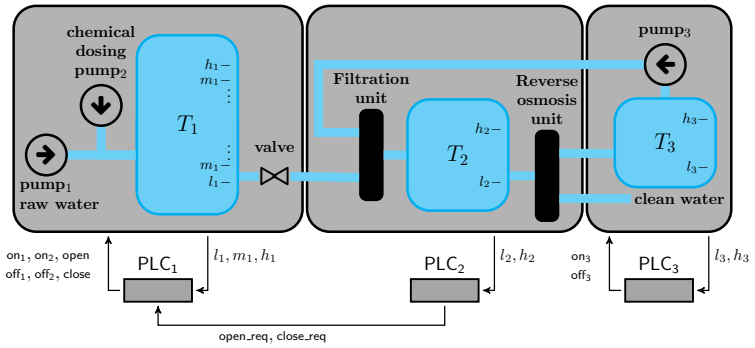



**Figure 7.1:** A simplified Industrial Water Treatment System.

stage, raw water is *chemically dosed* and pumped in a tank $T_1$, via two pumps $pump_1$ and $pump_2$. A *valve* connects $T_1$ with a *filtration unit* that releases the treated water in a second tank $T_2$. Here, we assume that the flow of the incoming water in $T_1$ is greater than the outgoing flow passing through the valve. The water in $T_2$ flows into a *reverse osmosis unit* to reduce inorganic impurities. In the last stage, the water coming from the reverse osmosis unit is either distributed as clean water, if required standards are met, or stored in a backwash tank $T_3$ and then pumped back, via a pump $pump_3$, to the filtration unit. Here, we assume that tank $T_2$ is large enough to receive the whole content of tank $T_3$ at any moment.

The SWaT system has been used to provide a dataset containing physical and network data recorded during 11 days of activity [64]. Part of this dataset contains information about the execution of the system in isolation, while a second part records the effects on the system when exposed to different kinds of cyber-physical attacks. Thus, for instance, (i) *drops* of commands to activate $pump_2$ may affect the quality of the water, as they would affect the correct functioning of the chemical dosing pump; (ii) *injections* of commands to close the valve between $T_1$ and $T_2$, may give rise to an overflow of tank $T_1$ if this tank is full; (iii) *integrity attacks* on the signals coming from the sensor of the tank $T_3$ may result in damages of the pump $pump_3$ if it is activated when $T_3$ is empty.

Each tank is controlled by its own PLC. The user programs of the three PLCs, expressed in terms of ladder logic, are given in Figure 7.2. In the following, we give their descriptions in `TCMC`.

Let us start with the code of the controller $PLC_1$ managing the tank $T_1$. Its definition is given in terms of two equations to deal with the case when the two pumps, $pump_1$ and $pump_2$, are both off and both on, respectively. Intuitively, when the pumps are off, the level of water in $T_1$ drops until it reaches its low level (event $l_1$); when this happens both pumps are turned on and they remain so until the tank is refilled,

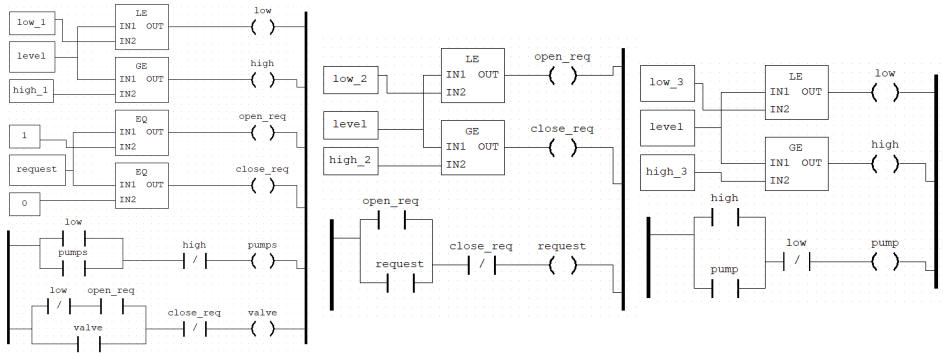



**Figure 7.2:** Ladder logics of the three PLCs controlling the system in Figure 7.1.

reaching its high level (event $h_1$). Formally,

$$
\begin{aligned}
P_1^{\mathsf{off}} = \mathsf{tick}.\big(\lfloor l_1.\overline{\mathsf{on}_1}.\overline{\mathsf{on}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{on}} \\
+ m_1.\lfloor \mathsf{open\_req}.\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{open}}.\mathsf{end}.P_1^{\mathsf{off}} + \mathsf{close\_req}.\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}} \rfloor \big(\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}}\big) \\
+ h_1.\lfloor \mathsf{open\_req}.\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{open}}.\mathsf{end}.P_1^{\mathsf{off}} + \mathsf{close\_req}.\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}} \rfloor \big(\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}}\big) \\
\rfloor \big(\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}}\big)\big) \\
P_1^{\mathsf{on}} = \mathsf{tick}.\big(\lfloor l_1.\overline{\mathsf{on}_1}.\overline{\mathsf{on}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{on}} \\
+ m_1.\lfloor \mathsf{open\_req}.\overline{\mathsf{on}_1}.\overline{\mathsf{on}_2}.\overline{\mathsf{open}}.\mathsf{end}.P_1^{\mathsf{on}} + \mathsf{close\_req}.\overline{\mathsf{on}_1}.\overline{\mathsf{on}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{on}} \rfloor \big(\overline{\mathsf{on}_1}.\overline{\mathsf{on}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{on}}\big) \\
+ h_1.\lfloor \mathsf{open\_req}.\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{open}}.\mathsf{end}.P_1^{\mathsf{off}} + \mathsf{close\_req}.\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}} \rfloor \big(\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{off}}\big) \\
\rfloor \big(\overline{\mathsf{off}_1}.\overline{\mathsf{off}_2}.\overline{\mathsf{close}}.\mathsf{end}.P_1^{\mathsf{on}}\big)\big)
\end{aligned}
$$

Thus, for instance, when the pumps are off the $\mathrm{PLC}_1$ waits for one time slot (to get stable sensor signals) and then checks the water level of the tank $T_1$, distinguishing between three possible states. If $T_1$ reaches its low level (signal $l_1$) then the pumps are turned on (commands $\overline{\mathsf{on}_1}$ and $\overline{\mathsf{on}_2}$) and the valve is closed (command $\mathsf{open\_req}$). Otherwise, if the tank $T_1$ is at some intermediate level between low and high (signal $m_1$) then $\mathrm{PLC}_1$ listens for requests arriving from $\mathrm{PLC}_2$ to open/close the valve. Precisely, if the PLC gets an $\mathsf{open\_req}$ request then it opens the valve, letting the water flow from $T_1$ to $T_2$, otherwise, if it gets a $\mathsf{close\_req}$ request then it closes the valve; in both cases the pumps remain off. If the level of the tank is high (signal $h_1$) then the requests of water coming from $\mathrm{PLC}_2$ are served as before, but the two pumps are eventually turned off (commands $\overline{\mathsf{off}_1}$ and $\overline{\mathsf{off}_2}$).

$\mathrm{PLC}_2$ manages the water level of tank $T_2$. Its code consists of the two equations to model the filling (state $\uparrow$) and the emptying (state $\downarrow$) of the tank. Formally,

$$
\begin{aligned}
P_2^{\uparrow} &= \mathsf{tick}.\big(\lfloor l_2.\lfloor \overline{\mathsf{open\_req}}.\mathsf{end}.P_2^{\uparrow}\rfloor \mathsf{end}.P_2^{\uparrow} + m_2.\lfloor \overline{\mathsf{open\_req}}.\mathsf{end}.P_2^{\uparrow}\rfloor \mathsf{end}.P_2^{\uparrow} + h_2.\lfloor \overline{\mathsf{close\_req}}.\mathsf{end}.P_2^{\downarrow}\rfloor \mathsf{end}.P_2^{\uparrow}\rfloor \mathsf{end}.P_2^{\uparrow}\big) \\
P_2^{\downarrow} &= \mathsf{tick}.\big(\lfloor l_2.\lfloor \overline{\mathsf{open\_req}}.\mathsf{end}.P_2^{\uparrow}\rfloor \mathsf{end}.P_2^{\downarrow} + m_2.\lfloor \overline{\mathsf{close\_req}}.\mathsf{end}.P_2^{\downarrow}\rfloor \mathsf{end}.P_2^{\downarrow} + h_2.\lfloor \overline{\mathsf{close\_req}}.\mathsf{end}.P_2^{\downarrow}\rfloor \mathsf{end}.P_2^{\downarrow}\rfloor \mathsf{end}.P_2^{\downarrow}\big)
\end{aligned}
$$

Here, after one time slot, the level of $T_2$ is checked. If the level is low (signal $l_2$) then $\mathrm{PLC}_2$ sends a request to $\mathrm{PLC}_1$, via the channel $\mathsf{open\_req}$, to open the valve that lets the water flow from $T_1$ to $T_2$, and then returns. Otherwise, if the level of tank $T_2$ is high (signal $h_2$) then $\mathrm{PLC}_2$ asks $\mathrm{PLC}_1$ to close the valve, via the channel $\mathsf{close\_req}$, and

then returns. Finally, if the tank $T_2$ is at some intermediate level between $l_2$ and $h_2$ (signal $m_2$) then the valve remains open (respectively, closed) when the tank is refilling (respectively, emptying).

Finally, $\text{PLC}_3$ manages the water level of the backwash tank $T_3$. Its code consists of two equations to deal with the case when the pump $pump_3$ is off and on, respectively. Formally,

$$P_3^{\mathsf{off}} = \mathsf{tick}.(\lfloor l_3.\overline{\mathsf{off}_3}.\mathsf{end}.P_3^{\mathsf{off}} + m_3.\overline{\mathsf{off}_3}.\mathsf{end}.P_3^{\mathsf{off}} + h_3.\overline{\mathsf{on}_3}.\mathsf{end}.P_3^{\mathsf{on}} \rfloor (\mathsf{off}_3.\mathsf{end}.P_3^{\mathsf{off}}))$$
$$P_3^{\mathsf{on}} = \mathsf{tick}.(\lfloor l_3.\overline{\mathsf{off}_3}.\mathsf{end}.P_3^{\mathsf{off}} + m_3.\overline{\mathsf{on}_3}.\mathsf{end}.P_3^{\mathsf{on}} + h_3.\overline{\mathsf{on}_3}.\mathsf{end}.P_3^{\mathsf{on}} \rfloor (\mathsf{off}_3.\mathsf{end}.P_3^{\mathsf{off}}))$$

Here, after one time slot, the level of tank $T_3$ is checked. If the level is low (signal $l_3$) then $\text{PLC}_3$ turns off the pump $pump_3$ (command $\overline{\mathsf{off}_3}$), and then returns. Otherwise, if the level of $T_3$ is high (signal $h_3$) then the pump is turned on (command $\overline{\mathsf{on}_3}$) until the whole content of $T_3$ is pumped back into the filtration unit of $T_2$.

## 7.3 A simple language for controllers' timed properties

In this section, we provide a simple description language to express *correctness properties* that we may wish to enforce at runtime in our controllers. As discussed in the Introduction, we resort to (a sub-class of) *regular properties*, the logical counterpart of regular expressions, as they allow us to express interesting classes of properties referring to one or more scan cycles of a controller.

### 7.3.1 Syntax and semantics

The proposed language distinguishes between two kinds of properties: (i) *global properties*, $e \in \mathbb{Prop}\mathbb{G}$, to express general controllers' execution traces; (ii) *local properties*, $p \in \mathbb{Prop}\mathbb{L}$, to express traces confined to a finite number of consecutive scan cycles. The two families of properties are formalised via the following regular grammar:

$$\begin{array}{lll} e \in \mathbb{Prop}\mathbb{G} & ::= & p^* \mid e_1 \cap e_2 \\ p \in \mathbb{Prop}\mathbb{L} & ::= & \epsilon \mid p_1; p_2 \mid \cup_{i \in I} \pi_i.p_i \mid p_1 \cap p_2 \end{array}$$

where $\pi_i \in \mathsf{Events} \triangleq \mathsf{Sens} \cup \overline{\mathsf{Act}} \cup \mathsf{Chn}^* \cup \{\mathsf{tick}\} \cup \{\mathsf{end}\}$ denote *atomic properties*, called *events*, that may occur as prefix of a property. With an abuse of notation, we use the symbol $\epsilon$ to denote both the *empty property* and the *empty trace*.

The *semantics* of our logic is naturally defined in terms of sets of execution traces which satisfy a given property; its formal definition is given in Table 7.3.

However, the syntax of our logic is a bit too permissive with respect to our intentions, as it allows us to describe partial scan cycles, *i.e.*, cycles that have not completed.

$$
\begin{array}{lcl}
[\![p^*]\!] & \triangleq & \{\epsilon\} \cup \bigcup_{n\in\mathbb{N}^+}\{t \mid t = t_1 \cdot \ldots \cdot t_n, \text{ with } t_i \in [\![p]\!], \text{ for } 1 \leq i \leq n\} \\
[\![e_1 \cap e_2]\!] & \triangleq & \{t \mid t \in [\![e_1]\!] \text{ and } t \in [\![e_2]\!]\} \\
[\![\epsilon]\!] & \triangleq & \{\epsilon\} \\
[\![p_1 \cap p_2]\!] & \triangleq & \{t \mid t \in [\![p_1]\!] \text{ and } t \in [\![p_2]\!]\} \\
[\![p_1; p_2]\!] & \triangleq & \{t \mid t = t_1 \cdot t_2, \text{ with } t_1 \in [\![p_1]\!] \text{ and } t_2 \in [\![p_2]\!]\} \\
[\![\bigcup_{i\in I} \pi_i.p_i]\!] & \triangleq & \bigcup_{i\in I}\{t \mid t = \pi_i \cdot t', \text{ with } t' \in [\![p_i]\!]\}
\end{array}
$$

**Table 7.3:** Trace semantics of our regular properties.

Thus, we restrict ourselves to considering properties which builds on top of local properties associated to *complete scan cycles*, *i.e.*, scan cycles whose last action is an $\mathsf{end}$-action. Formally,

**Definition 4.** Well-formed properties *are defined as follows:*

- *the local property* $\mathsf{end}.\epsilon$ *is well formed;*
- *a local property of the form* $p_1; p_2$ *is well formed if* $p_2$ *is well formed;*
- *a local property of the form* $p_1 \cap p_2$ *is well formed if both* $p_1$ *and* $p_2$ *are well formed;*
- *a local property of the form* $\cup_{i\in I}\pi_i.p_i$ *is well formed if either* $\pi_i.p_i = \mathsf{end}.\epsilon$ *or* $p_i$ *is well formed, for any* $i \in I$*;*
- *a global property* $p^*$ *is well formed if* $p$ *is well-formed;*
- *a global property* $e_1 \cap e_2$ *is well-formed if both* $e_1$ *and* $e_2$ *are well-formed.*

In the rest of the chapter, we always assume to work with well-formed properties. Moreover, we adopt the following notations and/or abbreviations on properties.

**Notation 2.** *We omit trailing empty properties, writing* $\pi$ *instead of* $\pi.\epsilon$*. For* $k > 0$*, we write* $\pi^k.p$ *as a shorthand for* $\pi.\pi...\pi.p$*, where prefix* $\pi$ *appears* $k$ *consecutive times. Given a local property* $p$ *we write* $\mathsf{events}(p) \subseteq \mathsf{Events}$ *to denote the set of events occurring in* $p$*; similarly, we write* $\mathsf{events}(e) \subseteq \mathsf{Events}$ *to denote the set of events occurring in a global property* $e \in \mathbb{P}\mathbb{r}\mathbb{o}\mathbb{p}\mathbb{G}$*. Given a set of events* $\mathcal{A} \subseteq \mathsf{Events}$ *and a local property* $p$*, we use* $\mathcal{A}$ *itself as an abbreviation for the property* $\cup_{\pi\in\mathcal{A}}\pi.\epsilon$*, and* $\mathcal{A}.p$ *as an abbreviation for the property* $\cup_{\pi\in\mathcal{A}}\pi.p$*. Given a set of events* $\mathcal{A}$*, with* $\mathsf{end} \notin \mathcal{A}$*, we write* $\mathcal{A}^{\leq k}$*, for* $k \geq 0$*, to denote the well-formed property defined as follows: (i)* $\mathcal{A}^{\leq 0} \triangleq \mathsf{end}$*; (ii)* $\mathcal{A}^{\leq k} \triangleq \mathsf{end} \cup \mathcal{A}.\mathcal{A}^{\leq k-1}$*, for* $k > 0$*. Thus, the property* $\mathcal{A}^{\leq k}$ *captures all possible sequences of events of* $\mathcal{A}$ *whose length is at most* $k$*, for* $k \in \mathbb{N}$*. We write* $\mathsf{PEvents}$ *to denote the set of pure events,* i.e., $\mathsf{Events} \setminus \{\mathsf{end}\}$*. Finally, we write* $\mathsf{PUEvents}$ *to denote the set of pure untimed events,* i.e., $\mathsf{Events} \setminus \{\mathsf{end}, \mathsf{tick}\}$.

### 7.3.2 Modularity: from simple to complex properties

From our simple language of regular properties we derive a wide family of correctness properties can be combined in a modular fashion to prescribe precise controller behaviours.

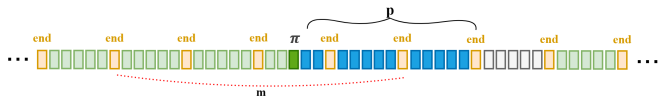



**Figure 7.3:** A trace satisfying a persistent conditional property $\mathrm{PCnd}_m(\pi, p)$.

For the sake of simplicity, both local and global properties presented in this section prescribe the behaviour of controllers whose initial sleeping phase is one tick long; these properties can be easily generalised to deal with $k$-tick sleeping phases: (i) $\mathsf{tick}^{k-1}.p$, for a local property $p$, and (ii) $(\mathsf{tick}^{k-1}.p)^*$, for a global property $p^*$. Furthermore, we assume a maximum number of actions, written maxa, that may occur within a single scan cycle of our controllers.

### 7.3.3 Local properties

As already said, local properties describe execution traces which are limited to a finite number of scan cycles. Let us present a number of significant local properties that can be expressed in our language of regular properties.

#### Basic properties

They prescribe conditional, eventual and persistent behaviours.

**Conditional** These properties says that when a (pure) untimed event $\pi$ occurs in the *current scan cycle* then some property $p$ should be satisfied. More generally, for $\pi_i \in \mathsf{PUEvents}$ and $p_i \in \mathbb{P}\mathrm{rop}\mathbb{L}$, we write $\mathrm{Case}(\cup_{i\in I}\{(\pi_i, p_i)\})$ to denote the property $q_k$, for $k = \mathsf{maxa}$, defined as follows:

- $q_k \triangleq \mathsf{end} \cup \bigcup_{i\in I} \pi_i.p_i \cup (\mathsf{PEvents} \setminus \bigcup_{i\in I}\{\pi_i\}).q_{k-1}$, for $0 < k \leq \mathsf{maxa}$
- $q_0 \triangleq \mathsf{end}$.

When there is only one triggering event $\pi \in \mathsf{PUEvents}$ and one associated local property $p \in \mathbb{P}\mathrm{rop}\mathbb{L}$, we have a simple conditional property defined as follow: $\mathrm{Cnd}(\pi, p) \triangleq \mathrm{Case}(\{(\pi, p)\})$.

Conditional properties $\mathrm{Cnd}(\pi, p)$ define a cause-effect relation in which the triggering event $\pi$ is searched in the current scan cycle; one may think of a more general property $\mathrm{PCnd}_m(\pi, p)$, in which the cause-effect relation *persists* for $m > 0$ consecutive scan cycles, *i.e.*, the search for the triggering event $\pi$ continues for at most $m$ consecutive scan cycles. Of course, the triggered local property $p$ may span over a finite number of scan cycles (see Figure 7.3). Formally, we write $\mathrm{PCnd}_m(\pi, p)$, for $\pi \in \mathsf{PUEvents}$, $p \in \mathbb{P}\mathrm{rop}\mathbb{L}$ and $m > 0$, for the property $q^m_{\mathsf{maxa}}$ defined as follows:

- $q^h_k \triangleq \mathsf{end}.q^{h-1}_{\mathsf{maxa}} \cup \pi.p \cup (\mathsf{PEvents}\setminus\{\pi\}).q^h_{k-1}$, for $1 < h \leq m$ and $0 < k \leq \mathsf{maxa}$
- $q^h_0 \triangleq \mathsf{end}.q^{h-1}_{\mathsf{maxa}}$, for $1 < h \leq m$
- $q^1_k \triangleq \mathsf{end} \cup \pi.p \cup (\mathsf{PEvents}\setminus\{\pi\}).q^1_{k-1}$, for $0 < k \leq \mathsf{maxa}$

- $q_0^1 \triangleq \epsilon$.

Obviously, $\mathrm{Cnd}(\pi, p) = \mathrm{PCnd}_1(\pi, p)$.

**Bounded eventually** In this case, *an event $\pi$ must eventually occur within $m$ scan cycles*. Formally, for $\pi \in \mathsf{PUEvents}$ and $m > 0$, we write $\mathrm{BE}_m(\pi)$ to denote the property $q^m_{\mathsf{maxa}}$ defined as follows, for $1 < h \leq m$ and $0 < k \leq \mathsf{maxa}$:

- $q_k^h \triangleq \mathsf{end}.q^{h-1}_{\mathsf{maxa}} \cup \pi.\mathsf{PEvents}^{\leq k-1} \cup (\mathsf{PEvents} \backslash \{\pi\}).q^h_{k-1}$
- $q_0^h \triangleq \mathsf{end}.q^{h-1}_{\mathsf{maxa}}$
- $q_k^1 \triangleq \pi.\mathsf{PEvents}^{\leq k-1} \cup (\mathsf{PEvents} \backslash \{\pi\}).q^1_{k-1}$
- $q_0^1 \triangleq \pi.\mathsf{end}.$

**Bounded persistency** While in $\mathrm{BE}_m(\pi)$ the event $\pi$ must eventually occur within $m$ scan cycles, bounded persistency prescribes that an event $\pi$ *must occur in all subsequent $m$ scan cycles*. Formally, for $\pi \in \mathsf{PUEvents}$ and $m > 0$, we write $\mathrm{BP}_m(\pi)$ to denote the property $q^m_{\mathsf{maxa}}$ defined as follows:

- $q_k^h \triangleq \pi.\mathsf{PEvents}^{\leq k-1}; q^{h-1}_{\mathsf{maxa}} \cup (\mathsf{PEvents} \backslash \{\pi\}).q^h_{k-1}$, for $1 < h \leq m$ and $0 < k \leq \mathsf{maxa}$
- $q_0^h \triangleq \pi.\mathsf{end}.q^{h-1}_{\mathsf{maxa}}$, for $1 < h \leq m$
- $q_k^1 \triangleq \pi.\mathsf{PEvents}^{\leq k-1} \cup (\mathsf{PEvents} \backslash \{\pi\}).q^1_{k-1}$, for $0 < k \leq \mathsf{maxa}$
- $q_0^1 \triangleq \pi.\mathsf{end}.$

**Bounded absence** The negative counterpart of bounded persistency is bounded absence. This property says that *an event $\pi$ must not appear in all subsequent $m$ scan cycles*. Formally, for $\pi \in \mathsf{PUEvents}$ and $m > 0$, we write $\mathrm{BA}_m(\pi)$ to denote the property $q_m$ defined as follows:

- $q_h \triangleq (\mathsf{PEvents} \backslash \{\pi\})^{\leq \mathsf{maxa}}; q_{h-1}$, for $0 < h \leq m$
- $q_0 \triangleq \epsilon$.

### Compound conditional properties

The properties above can be combined together to express more detailed PLC behaviours. Let us see a few examples with the help of the use case of Section 7.2.

**Conditional bounded eventually** According to this property, if a triggering event $\pi_1$ occurs then a second event $\pi_2$ must eventually occur between the $m$-th and the $n$-th scan cycle, with $1 \leq m \leq n$. Formally, for $\pi_1, \pi_2 \in \mathsf{PUEvents}$ and $1 \leq m \leq n$, we define $\mathrm{CBE}_{[m,n]}(\pi_1, \pi_2)$ as follows:

$$\mathrm{CBE}_{[m,n]}(\pi_1, \pi_2) \triangleq \mathrm{Cnd}(\pi_1, (\mathsf{PEvents}^{\leq \mathsf{maxa}})^{m-1}; \mathrm{BE}_{n-m+1}(\pi_2)).$$

Intuitively, if the event $\pi_1$ occurs then the event $\pi_2$ must eventually occur between the scan cycles $m$ and $n$. In case we would wish that $\pi_2$ should not occur before the $m$-th scan cycle, then the property would become: $\text{Cnd}(\pi_1, \text{BA}_{m-1}(\pi_2); \text{BE}_{n-m+1}(\pi_2))$.

As an example, we might enforce a conditional bounded eventually property in $\text{PLC}_1$ of our use case in Section 7.2 to prevent water overflow in the tank $T_2$ due to a misuse of the valve connecting the tanks $T_1$ and $T_2$. Assume that $z \in \mathbb{N}$ is the time (expressed in scan cycles) required to overflow the tank $T_2$ when the valve is open and the level of tank $T_2$ is low. We might consider to enforce a property of the form $\text{CBE}_{[1,w]}(\mathsf{open\_req}, \overline{\mathsf{close}})$, with $w << z$, saying that if $\text{PLC}_1$ receives a request to open the valve, then the valve will be eventually closed within at most $w$ scan cycles (including the current one). This will ensure that if a water request coming from $\text{PLC}_2$ is received by $\text{PLC}_1$ then the valve controlling the flow from $T_1$ to $T_2$ will remain open for at most $w$ scan cycles, with $w << z$, preventing the overflow of $T_2$.

**Conditional bounded persistency** Another possibility is to combine conditional with bounded persistency to prescribe that if a triggering event $\pi_1$ occurs then the event $\pi_2$ must occur in the $m$-th scan cycle and in all subsequent $n - m$ scan cycles, for $1 \leq m \leq n$. Formally, for $\pi_1, \pi_2 \in \mathsf{PUEvents}$ and $1 \leq m \leq n$, we write $\text{CBP}_{[m,n]}(\pi_1, \pi_2)$ to denote the property defined as:

$$\text{CBP}_{[m,n]}(\pi_1, \pi_2) \triangleq \text{Cnd}(\pi_1, (\mathsf{PEvents}^{\leq \mathsf{maxa}})^{m-1}; \text{BP}_{n-m+1}(\pi_2)).$$

As an example, we might enforce a conditional bounded persistency property in $\text{PLC}_3$ of our use case in Section 7.2 to prevent damages of $pump_3$ due to lack of water in tank $T_3$. Assume that $z \in \mathbb{N}$ is the minimum time (in terms of scan cycles) required to fill $T_3$, *i.e.*, to pass from level $l_3$ to level $h_3$, when $pump_3$ is off. We might consider to enforce a property of the form $\text{CBP}_{[1,z]}(l_3, \overline{\mathsf{off}_3})$, to prescribe that if the tank reaches its low level (event $l_3$) then $pump_3$ must remain off (event $\overline{\mathsf{off}_3}$) for $z$ consecutive scan cycles. This will ensure enough water in tank $T_3$ to prevent damages on $pump_3$.

Notice that all previous properties have a single triggering event $\pi_1$; in order to deal with multiple triggering events it is enough to replace the conditional operator with the case construct.

**Conditional bounded absence (also called Absence timed [58])** Finally, we might consider to combine conditional with bounded absence to formalise a property saying that if a triggering event $\pi_1$ occurs then another event $\pi_2$ must not occur in the $m$-th scan cycle and in all subsequent $n - m$ scan cycles, with $1 \leq m \leq n$. Formally, for $\pi_1, \pi_2 \in \mathsf{PUEvents}$ and $1 \leq m \leq n$, we write $\text{CBA}_{[m,n]}(\pi_1, \pi_2)$ to denote the property defined as follows:

$$\text{CBA}_{[m,n]}(\pi_1, \pi_2) \triangleq \text{Cnd}(\pi_1, (\mathsf{PEvents}^{\leq \mathsf{maxa}})^{m-1}; \text{BA}_{n-m+1}(\pi_2)).$$

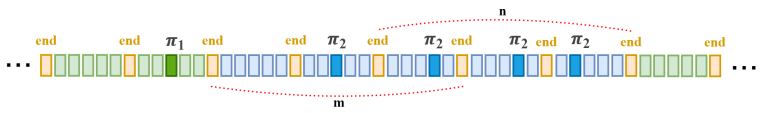



**Figure 7.4:** A trace satisfying a minimum duration property $\mathrm{MinD}(\pi_1, \pi_2, m, n)$, for $m = n = 3$.

Intuitively, if the triggering event $\pi_1$ occurs then the event $\pi_2$ must not occur in the time interval between the $m$-th and the $n$-th scan cycle.

As an example, we might enforce a conditional bounded absence property in $\mathrm{PLC}_2$ of our use case in Section 7.2 to prevent water overflow in the tank $T_2$ due to a misuse of the valve connecting the tanks $T_1$ and $T_2$. Assume that $z \in \mathbb{N}$ is the time (expressed in scan cycles) required to empty the tank $T_2$ when the valve is closed and the tank $T_2$ reaches its high level $h_2$. Then, we might consider to enforce a property of the form $\mathrm{CBA}_{[1,w]}(h_2, \overline{\mathsf{open\_req}})$, for $w < z$, to prescribe that if the tank reaches its high level (event $h_2$) then $\mathrm{PLC}_2$ may not send a requests to open the valve (event $\overline{\mathsf{open\_req}}$) for the subsequent $w$ scan cycles. This ensures us that when $T_2$ reaches its high level then it will not ask for incoming water for at least $w$ scan cycles, so preventing tank overflow.

### Compound persistent conditional properties

Now, we formalise in our language of regular properties a number of correctness properties used by Frehse et al. for the verification of hybrid systems [58]. More precisely, we formalise bounded versions of their properties.

**Bounded minimum duration** When a triggering event $\pi_1$ occurs, if a second event $\pi_2$ occurs within $m$ scan cycles then this event *must* appear in *at least* all subsequent $n$ scan cycles (see Figure 7.4). Formally, we can express this property as follows:

$$\mathrm{MinD}(\pi_1, \pi_2, m, n) \triangleq \mathrm{Cnd}(\pi_1, \mathrm{PCnd}_m(\pi_2, \mathrm{BP}_n(\pi_2))).$$

Note that the property $\mathrm{MinD}(\pi_1, \pi_2, m, n)$ is satisfied each time $\mathrm{CBP}_{[m,m+n]}(\pi_1, \pi_2)$ is. The vice versa does not hold as in $\mathrm{CBP}_{[m,m+n]}(\pi_1, \pi_2)$ the event $\pi_2$ is required to occur in the whole time interval $[m, m+n]$, while, according to $\mathrm{MinD}(\pi_1, \pi_2, m, n)$, the event $\pi_2$ might not occur at all.

**Bounded maximum duration** When an event $\pi_1$ occurs, if a second event $\pi_2$ occurs within $m$ scan cycles then the same event $\pi_2$ *may* occur in *at most* all subsequent $n$ scan cycles. Formally, we can represent this property as follows:

$$\mathrm{MaxD}(\pi_1, \pi_2, m, n) \triangleq \mathrm{Cnd}(\pi_1, \mathrm{PCnd}_m(\pi_2, (\mathsf{PEvents}^{\leq \mathsf{maxa}})^n; \mathrm{BA}_1(\pi_2))).$$

It is worth mentioning here that the property $\mathrm{MaxD}(\pi_1, \pi_2, m, n)$ is satisfied each time the property $\mathrm{CBP}_{[m,m+n]}(\pi_1, \pi_2); \mathrm{BA}_1(\pi_2)$ is. Again, the vice versa does not hold.

**Bounded response** When an event $\pi_1$ occurs, if a second event $\pi_2$ occurs within $m$ scan cycles then a third event $\pi_3$ appears within $n$ scan cycles. Formally, we can express this property as follows:

$$\mathrm{BR}(\pi_1, \pi_2, \pi_3, m, n) \triangleq \mathrm{Cnd}(\pi_1, \mathrm{PCnd}_m(\pi_2, \mathrm{BE}_n(\pi_3))).$$

**Bounded invariance** Whenever an event $\pi_1$ occurs, if $\pi_2$ occurs within $m$ scan cycles then $\pi_3$ will persistently occur for at least $n$ scan cycles. Formally, we can express this property as follows:

$$\mathrm{BI}(\pi_1, \pi_2, \pi_3, m, n) \triangleq \mathrm{Cnd}(\pi_1, \mathrm{PCnd}_m(\pi_2, \mathrm{BP}_n(\pi_3))).$$

**Bounded mutual exclusion**

A different class of properties prescribes the possible occurrence of events $\pi_i \in \mathsf{PEvents}$, for $i \in I$, in mutual exclusion within $m$ consecutive scan cycles. Formally, for $\pi_i \in \mathsf{PUEvents}$, $i \in I$ and $m \geq 1$, we write $\mathrm{BME}_m(\bigcup_{i\in I}\{\pi_i\})$, for the property $q^m_{\mathsf{maxa}}$ defined as:

- $q^h_k \triangleq \mathsf{end}.q^{h-1}_{\mathsf{maxa}} \cup \bigcup_{i\in I} \pi_i.(\bigcap_{j\in I\setminus\{i\}} \mathrm{BA}_h(\pi_j)) \cup (\mathsf{PEvents}\setminus\bigcup_{i\in I}\{\pi_i\}).q^h_{k-1}$, for $1 < h \leq m$ and $0 < k \leq \mathsf{maxa}$
- $q^h_0 \triangleq \mathsf{end}.q^{h-1}_{\mathsf{maxa}}$, for $1 < h \leq m$
- $q^1_k \triangleq \mathsf{end} \cup \bigcup_{i\in I} \pi_i.(\bigcap_{j\in I\setminus\{i\}} \mathrm{BA}_1(\pi_j)) \cup (\mathsf{PEvents}\setminus\bigcup_{i\in I}\{\pi_i\}).q^1_{k-1}$, for $0 < k \leq \mathsf{maxa}$
- $q^1_0 \triangleq \epsilon$.

As an example, we might enforce a bounded mutual exclusion property in the $\mathrm{PLC}_1$ of our use case of Section 7.2 to prevent chattering of the valve, *i.e.*, rapid opening and closing which may cause mechanical failures on the long run. In particular, we might consider to enforce a property of the form $\mathrm{BME}_3(\{\overline{\mathsf{open}}, \overline{\mathsf{close}}\})$ saying that within 3 consecutive scan cycles the opening and the closing of the valve (events $\overline{\mathsf{open}}$ and $\overline{\mathsf{close}}$, respectively) may only occur in mutual exclusion.

In Table 7.4, we summarise all local properties represented and discussed in this section.

### 7.3.4 Global properties

As expected, the previously described local properties become global ones by applying the Kleene-operator $*$. Once in this form, we can put these properties in conjunction between them. Here, we show two global properties, the first one is built top of conditional bounded persistency properties and the second one is built on top of a conditional bounded eventually property.

| | |
|---|---|
| Case: | if $\pi_i$ occurs then $p_i$ should be satisfied, for $i \in I$ |
| Persistent conditional: | for $m$ scan cycles, if $\pi$ occurs then $p$ should be satisfied |
| Bounded eventually: | event $\pi$ must eventually occur within $m$ scan cycles |
| Bounded persistency: | event $\pi$ must occur in all subsequent $m$ scan cycles |
| Bounded absence: | even $\pi$ must not occur in all subsequent $m$ scan cycles |
| Conditional bounded eventually: | if $\pi_1$ occurs then $\pi_2$ must eventually occur in the scan cycles $[m, n]$ |
| Conditional bounded persistency: | if $\pi_1$ occurs then $\pi_2$ must occur in all scan cycles of $[m, n]$ |
| Conditional bounded absence: | if $\pi_1$ occurs then $\pi_2$ must not occur in all scan cycles of $[m, n]$ |
| (Bounded) Minimum duration: | when $\pi_1$, if $\pi_2$ in $[1, m]$ then $\pi_2$ persists for at least $n$ scan cycles |
| (Bounded) Maximum duration: | when $\pi_1$, if $\pi_2$ in $[1, m]$ then $\pi_2$ persists for at most $n$ scan cycles |
| Bounded response: | when $\pi_1$, if $\pi_2$ in $[1, m]$ them $\pi_3$ appears within $n$ scan cycles |
| Bounded invariance: | when $\pi_1$, if $\pi_2$ in $[1, m]$ then $\pi_3$ persists for at least $n$ scan cycles |
| Bounded mutual exclusion | events $\pi_i$ may only occur in mutual exclusion within $n$ scan cycles |

**Table 7.4:** Overview of local properties.

As a first example, we might consider a global property saying that whenever an event $\pi$ occurs then all events $\pi_i$ , for $i \in I$, must occur in the $m$-th scan cycle and in all subsequent $n - m$ scan cycles, for $1 \leq m \leq n$. Formally, for $\pi, \pi_i \in \mathsf{PUEvents}$, $i \in I$, and $1 \leq m \leq n$: $\bigcap_{i \in I}(\mathrm{CBP}_{[m,n]}(\pi, \pi_i))^*$.

We might enforce this kind of property in $\mathrm{PLC}_1$ of our use case of Section 7.2. Assume $z \in \mathbb{N}$ being the time (expressed in scan cycles) required to overflow the tank $T_1$ when the level of the tank $T_1$ is low and both pumps are on and the valve is closed. Then, the property would be $(\mathrm{CBP}_{[1,w]}(l_1, \overline{\mathsf{on}_1}))^* \cap (\mathrm{CBP}_{[1,w]}(l_1, \overline{\mathsf{on}_2}))^*$, with $w < z$, saying that if the tank $T_1$ reaches its low level (event $l_1$) then both $pump_1$ and $pump_2$ must be on (events $\overline{\mathsf{on}_1}$ and $\overline{\mathsf{on}_2}$) in all subsequent $w$ scan cycles, starting from the current one.

As a second example, we might consider a more involved global property relying on conditional bounded eventually, persistent conditional and bounded persistency. Basically, the property says that whenever an event $\pi_1$ occurs then a second event $\pi_2$ must eventually occur between the $m$-th scan cycle and the $n$-th scan cycle, with $1 \leq m \leq n$; moreover, it must occur for $d$ consecutive scan cycles, for $1 \leq d$ (see Figure 7.5). Formally, the property is the following:

$$\left(\mathrm{CBE}_{[m,n]}(\pi_1, \pi_2)\right)^* \cap \left(\mathrm{Cnd}(\pi_1, \mathrm{PCnd}_n(\pi_2, \mathsf{PEvents}^{\leq \mathsf{maxa}}; \mathrm{BP}_{d-1}(\pi_2)))\right)^*$$

for $\pi_1, \pi_2 \in \mathsf{PUEvents}$, with $1 \leq m \leq n$ and $d \geq 1$. Intuitively, $(\mathrm{CBE}_{[m,n]}(\pi_1, \pi_2))^*$ requires that when $\pi_1$ occurs the event $\pi_2$ must eventually occur between the $m$-th scan cycle and the $n$-th scan cycle. The remaining part of the property says if the event $\pi_2$ occurs within the $n$-th scan cycle (recall that $m \leq n$) then it must persist for $d$ scan cycles.

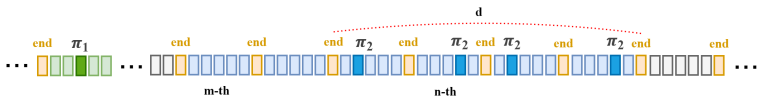



**Figure 7.5:** A trace satisfying the aforementioned property for some $m$, $n = m + 4$ and $d = 4$.

In this manner, we might strengthen the conditional bounded eventually property given in Section 7.3.3 for $\text{PLC}_1$ of our use case to prevent water overflow in the tank $T_2$. Let $z \in \mathbb{N}$ be the time (expressed in scan cycles) required to overflow the tank $T_2$ when the valve is open and the level of tank $T_2$ is low. The property is the following:

$$(\text{CBE}_{[1,w]}(\mathsf{open\_req}, \overline{\mathsf{close}}))^* \cap (\text{Cnd}(\mathsf{open\_req}, \text{PCnd}_w(\overline{\mathsf{close}}, \text{PEvents}^{\leq \mathsf{maxa}}; \text{BP}_{d-1}(\overline{\mathsf{close}}))))^*$$

where $w << z$, and $d \in \mathbb{N}$ is the time (expressed in scan cycles) required to release in $T_3$ the (maximum) quantity of water that the tank $T_2$ may accumulate in $w$ scan cycles. The first part of the property says that if $\text{PLC}_1$ receives a request to open the valve (event $\mathsf{open\_req}$) then the valve must be eventually closed (event $\overline{\mathsf{close}}$ must eventually occur) within at most $w$ scan cycles. The remaining part of the property says that when $\text{PLC}_1$ receives a request to open the valve (event $\mathsf{open\_req}$), if the valve gets closed (event $\overline{\mathsf{close}}$) within the $w$-th scan cycle, then it must remain closed for the $d$ consecutive scan cycles. Here, $d$ depends both on the maximum level of water reachable in $T_2$ in $w$ scan cycles and on the physical law governing the water flow from $T_2$ to $T_3$.

## 7.4 Monitor synthesis

In this section, we provide an algorithm to synthesise monitors from regular properties whose events are contained in (the set of events associated to) a fixed set $\mathcal{P}$ of observable actions. More precisely, given a global property $e \in \mathbb{P}\mathrm{r}\mathbb{o}\mathbb{p}\mathbb{G}$ the algorithm returns an edit automaton $\langle\!\|\, e \,\|\!\rangle^{\mathcal{P}} \in \mathbb{E}\mathrm{d}\mathrm{i}\mathrm{t}$ that is capable to enforce the property $e$ during the execution of a generic controller whose possible actions are confined to those in $\mathcal{P}$.

### 7.4.1 Synthesis algorithm

The synthesis algorithm is defined in Table 7.5 by induction on the structure of the global/local property given in input; as we distinguish global properties from local ones, we define our algorithm in two steps.

The monitor $\langle\!\|\, p^* \,\|\!\rangle^{\mathcal{P}}$ associated to a global property $p^*$ is an edit automaton defined via the recursive equation $\mathsf{X} = \langle\!\|\, p \,\|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, to recursively enforce the local property $p$. The monitor $\langle\!\|\, e_1 \cap e_2 \,\|\!\rangle^{\mathcal{P}}$ is given by the *cross product* between the edit automata $\langle\!\|\, e_1 \,\|\!\rangle^{\mathcal{P}}$ and $\langle\!\|\, e_2 \,\|\!\rangle^{\mathcal{P}}$, to accept only traces that satisfy both $e_1$ and $e_2$; the technical definition of the cross product $\text{Prod}^{\mathcal{P}}_{\mathsf{X}}(\mathsf{E}_1, \mathsf{E}_2)$ between two edit automata $\mathsf{E}_1$ and $\mathsf{E}_2$, with respect a process variable $\mathsf{X}$, is given in the appendix in Table 11.1.

The monitor $\langle\!\|\, p_1 \cap p_2 \,\|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ is given by the cross product of Table 11.1 between the edit automata $\langle\!\|\, p_1 \,\|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ and $\langle\!\|\, p_2 \,\|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$. The monitor $\langle\!\|\, p_1; p_2 \,\|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ is given by the sequential composition of the edit automata associated to the properties $p_1$ and $p_2$, respectively. Finally, the monitor associated to a union property $\cup_{i \in I} \pi_i.p_i$ does the following: (i) *allows* all actions associated to the events $\pi_i$, (ii) *inserts* the action associated to the

$$
\begin{array}{rcl}
\langle\!|\, p^* \,|\!\rangle^{\mathcal{P}} & \triangleq & \mathsf{X},\ \text{for } \mathsf{X} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} \\
\langle\!|\, e_1 \cap e_2 \,|\!\rangle^{\mathcal{P}} & \triangleq & \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, e_1 \,|\!\rangle^{\mathcal{P}}, \langle\!|\, e_2 \,|\!\rangle^{\mathcal{P}}),\ \mathsf{X} \text{ fresh} \\
\langle\!|\, \epsilon \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} & \triangleq & \mathsf{X} \\
\langle\!|\, p_1 \cap p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} & \triangleq & \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \\
\langle\!|\, p_1; p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} & \triangleq & \langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{Z}},\ \text{for } \mathsf{Z} = \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}},\ \mathsf{Z} \text{ fresh} \\
\langle\!|\, \bigcup_{i\in I} \pi_i.p_i \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} & \triangleq & \mathsf{Z},\ \text{for } \mathsf{Z} = \sum\limits_{i\in I} \pi_i.\langle\!|\, p_i \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} + \sum\limits_{\substack{i\in I \\ \pi_i \neq \mathsf{end}}} \pi_i \prec \mathsf{end}.\langle\!|\, p_i \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} + \sum\limits_{\alpha\in\mathcal{P}\setminus(\cup_{i\in I}\pi_i\cup\{\mathsf{tick},\mathsf{end}\})} {}^{-}\alpha.\mathsf{Z}
\end{array}
$$

**Table 7.5:** Monitor synthesis from properties in $\mathbb{P}\mathrm{rop}\mathbb{G}$ and $\mathbb{P}\mathrm{rop}\mathbb{L}$.

events $\pi_i$ whenever the controller is about to complete the scan cycle, i.e., to emit an end-action, and (iii) *suppresses* any other event.

**Remark 5.** *Notice that our synthesised monitors do not suppress neither* tick*-actions nor* end*-actions. Thus, if the monitor is aligned with the controller on the execution of an* end*-actions, then a new scan cycle is free to start; if this is not the case, the monitor yields some correct trace, without any involvement of the controller, to reach the completion of the current scan cycle.*

### 7.4.2 Enforcement properties

First of all, we calculate the complexity of the synthesis algorithm based on the number of occurrences of the operator $\cap$ in $e$ and the dimension of $e$, $\mathsf{dim}(e)$, *i.e.*, the number of all operators occurring in $e$ (for the definition of $\mathsf{dim}(e)$ the reader is referred to the appendix).

**Proposition 1** (Complexity)**.** *Let $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ be a global property and $\mathcal{P}$ be a set of actions such that $\mathsf{events}(e) \subseteq \mathcal{P}$. The complexity of the algorithm to synthesise $\langle\!|\, e \,|\!\rangle^{\mathcal{P}}$ is $\mathcal{O}(m^{k+1})$, with $m = \mathsf{dim}(e)$ and $k$ being the number of occurrences of the operator $\cap$ in $e$.*

In the following, we prove that the enforcement induced by our synthesised monitors enjoys the properties stated in the Introduction: *determinism preservation*, *transparency*, *soundness*, *deadlock-freedom*, *divergence-freedom*, and *compositionality*. In this section, with a small abuse of notation, given a set of observable actions $\mathcal{P}$, we will use $\mathcal{P}$ to denote also the set of the corresponding events.

As concerns determinism preservation, we prove that deterministic properties give rise to deterministic enforcements, *i.e.*, monitored controllers with a deterministic runtime behaviour [32].

**Definition 5** (Deterministic properties)**.** *A global property $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ is said to be deterministic if for any sub-term $\cup_{i\in I}\pi_i.p_i$ appearing in $e$, we have $\pi_k \neq \pi_h$, for any $k, h \in I$, $k \neq h$.*

Notice that all properties proposed in Section 7.3 are deterministic.

**Definition 6** (Semantically deterministic enforcement)**.** *Given an edit automaton* $\mathsf{E} \in \mathbb{E}\mathrm{dit}$ *and a controller* $P \in \mathbb{C}\mathrm{trl}$*. The monitored controller* $\mathsf{E} \bowtie P$ *is said to be* semantically deterministic *if for any trace* $t$ *and actions* $\alpha_1, \alpha_2$ *such that* $\mathsf{E} \bowtie P \xrightarrow{t} \mathsf{E}' \bowtie J'$ *and* $\mathsf{E}' \bowtie J' \xrightarrow{\alpha_1} \mathsf{E}_1 \bowtie J_1$ *and* $\mathsf{E}' \bowtie J' \xrightarrow{\alpha_2} \mathsf{E}_2 \bowtie J_2$ *and* $\mathsf{E}_1 \neq \mathsf{E}_2$*, it holds that* $\alpha_1 \neq \alpha_2$*.*

Thus, given a deterministic global property $e$, our synthesis algorithm returns a semantically deterministic enforcer. Formally,

**Proposition 2** (Deterministic preservation)**.** *Given a deterministic global property* $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ *over a set of events* $\mathcal{P}$*. For any controller* $P \in \mathbb{C}\mathrm{trl}$*, the monitored controller* $\langle\!| e |\!\rangle^{\mathcal{P}} \bowtie P$ *is a semantically deterministic.*

Let us move to the next property: *transparency*. Intuitively, the enforcement induced by a property $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ should not prevent any trace satisfying $e$ itself [102].

**Theorem 1** (Transparency)**.** *Let* $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ *be a global property,* $\mathcal{P}$ *be a set of observable actions such that* $\mathsf{events}(e) \subseteq \mathcal{P}$*, and* $P \in \mathbb{C}\mathrm{trl}$ *be a controller. Let* $t$ *be a trace of* $\mathsf{go} \bowtie P$*. If* $t \in [\![e]\!]$ *then* $t$ *is a trace of* $\langle\!| e |\!\rangle^{\mathcal{P}} \bowtie P$*.*

Another important property of our enforcement is *soundness* [102]. Intuitively, a controller under the scrutiny of a monitor $\langle\!| e |\!\rangle^{\mathcal{P}}$ should only yield execution traces which satisfy the enforced property $e$, *i.e.*, execution traces which are consistent with its semantics $[\![e]\!]$ (up to $\tau$-actions).

**Theorem 2** (Soundness)**.** *Let* $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ *be a global property,* $\mathcal{P}$ *be a set of observable actions such that* $\mathsf{events}(e) \subseteq \mathcal{P}$*, and* $P \in \mathbb{C}\mathrm{trl}$ *be a controller. If* $t$ *is a trace of the monitored controller* $\langle\!| e |\!\rangle^{\mathcal{P}} \bowtie P$ *then* $\hat{t}$ *is a prefix of some trace in* $[\![e]\!]$ *(see Notation 1 for the definition of the trace* $\hat{t}$*).*

Here, it is important to stress that in general soundness does not ensure deadlock-freedom of the monitored controller. That is, it may be possible that the enforcement of some property $e$ causes a deadlock of the controller $P$ under scrutiny. In particular, this may happen in our controllers only when the initial sleeping phase does not match the enforcing property. Intuitively, a local property will be called a $k$-sleeping property if it allows $k$ initial time instants of sleep. Formally,

**Definition 7.** *For* $k \in \mathbb{N}^+$*, we say that* $p \in \mathbb{P}\mathrm{rop}\mathbb{L}$ *is a* $k$-sleeping local property*, only if* $[\![p]\!] = \{t \mid t = t_1 \cdot \ldots \cdot t_n$*, for* $n > 0$*, s.t.* $t_i = \mathsf{tick}^k \cdot t_i' \cdot \mathsf{end}, \mathsf{end} \notin t_i'$*, and* $1 \leq i \leq n\}$*. We say that* $p^*$ *is a* $k$-sleeping global property *only if* $p$ *is, and* $e = e_1 \cap e_2$ *is* $k$*-sleeping only if both* $e_1, e_2$ *are* $k$*-sleeping.*

The enforcement of $k$-sleeping properties does not introduce deadlocks in $k$-sleeping controllers. This is because our synthesised monitors suppress all incorrect actions of the controller under scrutiny, driving it to the end of its scan cycle. Then, the controller remains in stand-by while the monitor yields a safe sequence of actions to mimic a safe completion of the current scan cycle.

**Theorem 3** (Deadlock-freedom)**.** *Let* $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ *be a* $k$*-sleeping global property, and* $\mathcal{P}$ *be a set of observable actions such that* $\mathsf{events}(e) \subseteq \mathcal{P}$*. Let* $P \in \mathbb{C}\mathrm{trl}$ *be a controller of the*

*form* $P = \mathsf{tick}^k.S$ *whose set of observable actions is contained in* $\mathcal{P}$*. Then,* $(\!|\, e \,|\!)^{\mathcal{P}} \bowtie P$ *does not deadlock.*

Another important key of our enforcement mechanism is *divergence-freedom* [52]. In practice, the enforcement does not introduce divergence: monitored controllers will always be able to complete their scan cycles by executing a finite number of actions. This is because we limit our enforcement to well-formed properties (Definition 4) which always terminates with an end event. In particular, the well-formedness of local properties ensures us that in a global property of the form $p^*$ the number of events within two subsequent end events is always finite.

**Theorem 4** (Divergence-freedom)**.** *Let* $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ *be a global property,* $\mathcal{P}$ *be a set of observable actions such that* $\mathsf{events}(e) \subseteq \mathcal{P}$*, and* $P \in \mathbb{C}\mathrm{trl}$ *be a controller. Then, there exists a* $k \in \mathbb{N}^+$ *such that whenever* $(\!|\, e \,|\!)^{\mathcal{P}} \bowtie P \xrightarrow{t} \mathsf{E} \bowtie J$*, if* $\mathsf{E} \bowtie J \xrightarrow{t'} \mathsf{E}' \bowtie J'$*, with* $|\ t' \ | \geq k$*, then* $\mathsf{end} \in t'$*.*

Notice that all properties seen up to now hold in *field networks* of controllers. This means that they are preserved when the controller under scrutiny is running in parallel with other controllers in the same field communications network. As an example, by an application of Theorems 1 and 2, we show how both transparency and soundness hold in field networks of controllers running in parallel. A similar result applies to the remaining properties.

**Corollary 1** (Compositionality)**.** *Let* $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ *be a global property and* $\mathcal{P}$ *be a set of observable actions, such that* $\mathsf{events}(e) \subseteq \mathcal{P}$*. Let* $P \in \mathbb{C}\mathrm{trl}$ *be a controller and* $N \in \mathbb{FN}\mathrm{et}$ *be a field network. If* $((\!|\, e \,|\!)^{\mathcal{P}} \bowtie P) \parallel N \xrightarrow{t} (\mathsf{E} \bowtie J) \parallel N'$*, for some* $t$*,* $\mathsf{E}$*,* $J$ *and* $N'$*, then*

- *whenever* $\mathsf{go} \bowtie P \xrightarrow{t'} \mathsf{go} \bowtie J$*, with* $t' \in [\![e]\!]$*, the trace* $t'$ *is a trace of* $(\!|\, e \,|\!)^{\mathcal{P}} \bowtie P$*;*
- *whenever* $(\!|\, e \,|\!)^{\mathcal{P}} \bowtie P \xrightarrow{t'} \mathsf{E} \bowtie J$ *the trace* $\widehat{t'}$ *is a prefix of some trace in* $[\![e]\!]$*.*

## 7.5 Summary

In this chapter we have defined a simple timed process calculus, based on Hennessy and Regan's *Timed Process Language* (TPL) [75], for specifying controllers, finite-state edit automata, and networks of communicating monitored controllers, Table 7.2. Then, we have defined a simple description language based on regular expression to express *timed correctness properties* that should hold upon completion of a finite number of scan cycles of the monitored controller, indeed, we have have focused only on well-formed property according to Definition 4. This language has allowed us to abstract over controllers implementations, focusing on general properties which may even be shared by completely different controllers. With such language, we have shown a wide class of interesting correctness properties for controllers in terms of our regular properties, Table 7.4. Furthermore, we have provided a *synthesis function*, $(\!|\, - \,|\!)$ that, given an alphabet $\mathcal{P}$ of observable actions (sensor readings, actuator commands, and inter-controller communications) and a deterministic regular property $e$ combining events of $\mathcal{P}$, returns

an edit automaton $\langle\!| e |\!\rangle^{\mathcal{P}}$, Table 7.5. The resulting enforcement mechanism will ensure the required features: transparency, soundness, determinism preservation, deadlock-freedom, divergence-freedom, mitigation and compositionality. Finally, in Section 7.2 we have proposed a non-trivial case study, taken from the context of industrial water treatment systems. There, we have also shown the user programs of the three PLCs, expressed in terms of ladder logic, Figure 7.2. Thus, in the next chapter, we will rely on this case study to show the effectiveness of our enforcement mechanism.

# Chapter 8

# Our enforcement mechanism at work

In this chapter, we propose an implementation of our enforcement mechanism in which monitors, running on *field-programmable gate arrays* (FPGAs) [150], enforce *open source PLCs* [1] [11], running on Raspberry Pi devices [60].

This chapter has the following structure. In Section 8.1, we argue why FPGAs are good candidates for implementing *secure proxies*. In Section 8.2, we describe how we implemented the enforcement of non-trivial safety properties on the use case of Section 7.2, where the physical plant was simulated in *Simulink* [110]. Finally, in Section 8.3, we test our implementation when the enforced PLCs are injected with five different malware aiming at causing three different physical perturbations: tank overflow, valve damage, and pump damage.

## 8.1 FPGAs as secure proxies for ICSs

Field-programmable gate arrays (FPGAs) are semiconductor devices that can be programmed to run specific applications. An FPGA consists of (configurational) logic blocks, routing channels and I/O blocks. The logic blocks can be configured to perform complex combinational functions and are further made up of transistor pairs, logic gates, lookup tables and multiplexers. The applications are written using hardware description languages, such as Verilog [142]. Thus, in order to execute an application on the FPGA, its Verilog code is converted into a sequence of bits, called *bitstream*, that is loaded into the FPGA.

FPGAs operate with a frequency of 100 MHz, thus, they introduce a negligible overhead in the whole behaviour of the PLCs which run with a frequency of 1 KHz [118]. Furthermore, FPGAs are assumed to be secure when the adversary does not have physical access to the device, *i.e.*, the bitstream cannot be compromised [80]. Recent FPGAs support remote updates of the bitstream by relying on authentication mechanisms to

[1] Compliant with the IEC 61,131−3 international standard.

prevent unauthorised uploads of malicious logic [80]. Nevertheless, as said in the Introduction and advocated by McLaughlin and Mohan [111, 114], any form of runtime reconfiguration should be prevented. Summarising, under the assumption that the adversary does not have physical access to the FPGA and she cannot do remote updates, FPGAs represent a good candidate for the implementation of secure enforcing proxies.

## 8.2 An implementation of the enforcement of the SWaT system

The proposed implementation adopts different approaches for plant, controllers and enforcers.

**Plant** The plant of the SWaT system is simulated in Simulink [110], a framework to model, simulate and analyse cyber-physical systems, widely adopted in industry and research. A Simulink model is given by *blocks* interconnected via *wires*. Our Simulink model contains blocks to simulate water tanks, actuators (*i.e.*, pumps and valves) and sensors (see Figure 8.1). In particular, water-tank blocks implement the differential equations that model the dynamics of the tanks according to the physical constraints obtained from [109, 64]. Actuation blocks receive commands from PLCs, whereas sensor blocks send measurements to PLCs. For simplicity, state changes of both pumps and valves do not occur instantaneously; they take 1 second. Finally, we ran our Simulink model on a laptop with 2.8 GHz Intel i7 7700 HQ, 16GB memory, and Linux Ubuntu 20.04 LTS OS.

**Controllers** Controllers use *OpenPLC* [11], a fully functional open source PLC capable of running user programs in all five IEC61131-3 defined languages [82]. Additionally, OpenPLC supports standard SCADA protocols, such as Modbus/TCP, DNP3 and Ethernet/IP. OpenPLC can run on a variety of hardware, from a simple Raspberry Pi to robust industrial boards. We installed OpenPLC on three Raspberry Pi 4 [131]; each instance runs one of the three ladder logics seen in Figure 7.2.

**Enforcers** As regards the enforcing monitors, we use three NetFPGA-CML development boards [154]. More precisely, we have implemented our synthesis algorithm in

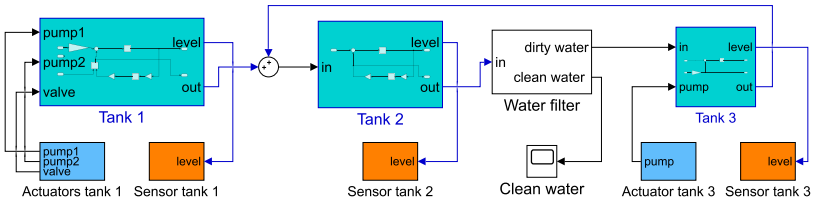



**Figure 8.1:** An implementation in Simulink of the plant of the SWaT system.

Python to return enforcers written in Verilog. The Verilog code is then compiled into a bitstream which runs inside the FPGAs.

Finally, the connection between the PLCs, the enforcers and the Simulink of the plant is realised via a wired UDP network (see Figure 8.2). The duration of each scan cycle is fixed to 100 milliseconds with no enforcement, and 101 milliseconds in the presence of enforcement. The ladder logics of the three PLCs, the synthesis algorithm in Python, the enforcers written in Verilog, and the Simulink simulations can be found at: https://bitbucket.org/formal_projects/runtime_enforcement.

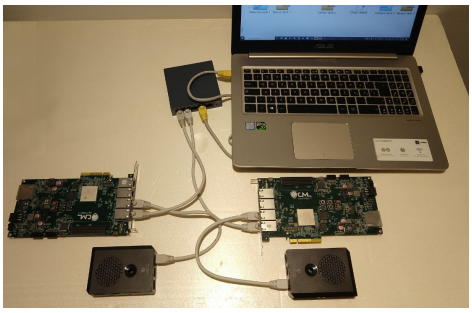


**Figure 8.2:** Some components of our implementation.

## 8.3 The enforced SWaT system under attack

In this section, we consider five different attacks targeting the PLCs of the SWaT system to achieve three possible malicious goals: (i) overflow the water tanks, (ii) damage of the valve, (iii) damage of the pumps. In order to simulate the injection of malware in the PLCs, we reinstall the original PLC ladder logics with compromised ones, containing some additional logic intended to disrupt the normal operations of the PLC [67]. In the following, we will discuss these attacks, grouped by goals, showing how the enforcement of specific properties mitigates the attacks by preserving the correct behaviour of the monitored PLCs.

**Tank overflow** Our *first attack* is a DoS attack targeting $\text{PLC}_1$ by dropping the commands to close the valve. In the left-hand side of Figure 8.3 we show a possible implementation of this attack in ladder logic. Basically, the malware remains silent for 500 seconds and then it sets true a malicious *drop* variable (highlighted in yellow). Once the variable *drop* becomes true, the *valve* variable is forced to be false (highlighted in red), thus preventing the closure of the valve.

In order to prevent attacks aiming at overflowing the tanks, we propose the following three enforcing properties, one for each PLC, respectively:

- $e_1 \triangleq (\text{CBP}_{[1,m]}(h_1, \overline{\text{off}_1}))^* \cap (\text{CBP}_{[1,m]}(h_1, \overline{\text{off}_2}))^*$, an intersection between two conditional bounded persistency properties to enforce $\text{PLC}_1$ to prevent water overflow in $T_1$. This property ensures that both pumps $pump_1$ and $pump_2$ are off, for

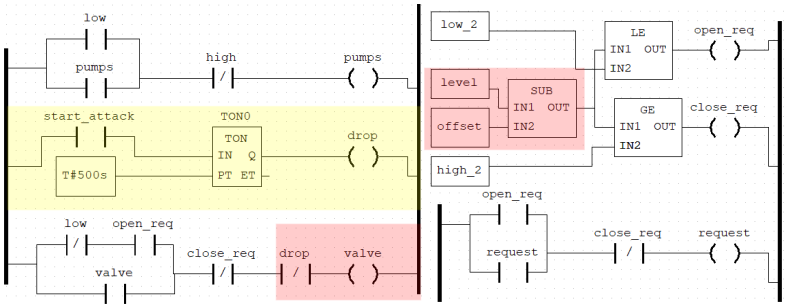



**Figure 8.3:** Tank overflow: Ladder Logic of the first (left) and the second attack (right).

$m$ consecutive scan cycles, when the level of $T_1$ is high (measurement $h_1$). Here, $m < n$ for $n \in \mathbb{N}$ is the number of scan cycles required to empty $T_1$ when its level is high, both pumps are off, and the valve is open.

- $e_2 \triangleq (\text{CBP}_{[1,u]}(h_2, \overline{\text{close\_req}}))^*$, a conditional bounded persistency property for $\text{PLC}_2$ ensuring that requests to close the valve (event $\overline{\text{close\_req}}$) are sent for $u$ consecutive scan cycles when the level of water in tank $T_2$ is high (measurement $h_2$). Here, $u < v$ for $v \in \mathbb{N}$ is the number of scan cycles required to empty the tank $T_2$ when the level is high and the valve is closed.
- $e_3 \triangleq (\text{CBP}_{[1,w]}(h_3, \overline{\text{on}_3}))^*$, a conditional bounded persistency property for $\text{PLC}_3$ to ensure that $pump_3$ is on for $w$ consecutive scan cycles when the level of water in tank $T_3$ is high (measurement $h_3$). Here, $w < z$ for $z \in \mathbb{N}$ is the time (expressed in scan cycles) required to empty the tank $T_3$ when the level is high and $pump_3$ is on.

In Figure 8.4 we show part of the Verilog code of the enforces associated to the properties $e_1, e_2$ and $e_3$, respectively.

Now, let us analyse the effectiveness of the enforcement induced by these three properties. For instance, in the upper graphs of Figure 8.5 we report the impact on the tanks $T_1$ and $T_2$ of the DoS attack previously described, when enforcing the three

```
...
case(state)
    INIT:
        if (level_1 >= high_1) begin
            state = ENF_PUMP_1;
        end
        else begin
            state = INIT;
        end
    ENF_PUMP_1:
        if (pump_1 == off) begin
            state = ENF_PUMP_2;
        end
        else begin
            pump_1 = off;
            state = ENF_PUMP_2;
        end
    ENF_PUMP_2:
        if (pump_2 == off) begin
            state = CYCLE_1;
        end
        else begin
            pump_2 = off;
            state = CYCLE_1;
        end
...
```

```
...
case(state)
    INIT:
        if (level_2 >= high_2) begin
            state = ENF_REQ;
        end
        else begin
            state = INIT;
        end
    ENF_REQ:
        if (request == close_req) begin
            state = CYCLE_1;
        end
        else begin
            request = close_req;
            state = CYCLE_1;
        end
    CYCLE_1:
        if (request == close_req) begin
            state = CYCLE_2;
        end
        else begin
            request = close_req;
            state = CYCLE_2;
        end
...
```

```
...
case(state)
    INIT:
        if (level_3 >= high_3) begin
            state = ENF_PUMP;
        end
        else begin
            state = INIT;
        end
    ENF_PUMP:
        if (pump_3 == on) begin
            state = CYCLE_1;
        end
        else begin
            pump_3 = on;
            state = CYCLE_1;
        end
    CYCLE_1:
        if (pump_3 == on) begin
            state = CYCLE_2;
        end
        else begin
            pump_3 = on;
            state = CYCLE_2;
        end
...
```

**Figure 8.4:** Verilog code of the enforcers of the three properties $e_1, e_2, e_3$.

properties $e_1, e_2$ and $e_3$ in the corresponding PLCs. Here, the red region denotes when the attack becomes active. As the reader may notice, despite repeated requests to close the valve coming from $PLC_2$, the compromised $PLC_1$ never closes the valve causing the overflow of tank $T_2$. So, the enforced property $e_1$ is not up the task.

In order to prevent this attack, we must guarantee that $PLC_1$ closes the valve when $PLC_2$ requests so. Thus, we should enforce in $PLC_1$ a more demanding property $e_1'$ defined as follows: $e_1 \cap \text{CBE}_{[1,1]}(\text{close\_req}, \overline{\text{close}})$. Basically, the last part of the property ensures that every request to close the valve is followed by an actual closure of the valve in the same scan cycle. The impact of the malware on $PLC_1$ when enforcing the properties $e_1', e_2, e_3$ is represented in the lower graphs of Figure 8.5. Now, the correct behaviour of $PLC_1$ is ensured, thus preventing the overflowing of the water tank $T_2$. In these graphs, the green highlighted regions denote when the monitor *detects* the attack and *mitigates* the activities of the compromised $PLC_1$. In particular, the monitor *inserts* the commands to close the valve on behalf of $PLC_1$ when $PLC_2$ sends requests to close the valve.

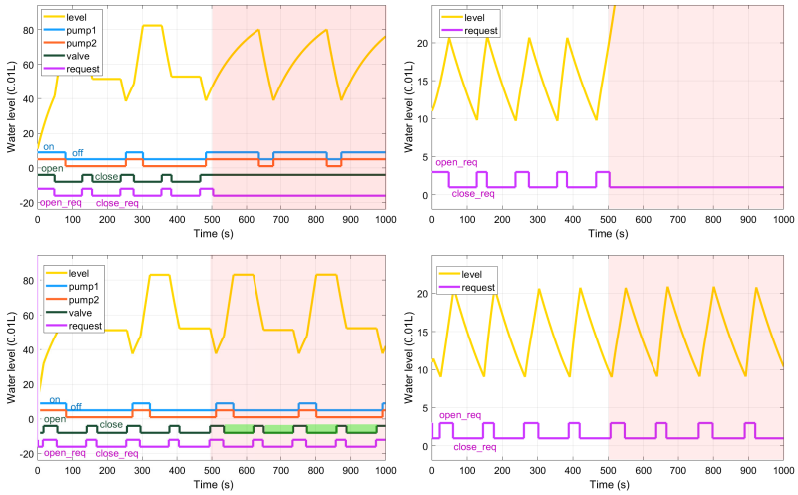



**Figure 8.5:** Tank overflow: DoS attack on $PLC_1$ when enforcing $e_1, e_2, e_3$ (up) and $e_1', e_2, e_3$ (down).

Having strengthened the enforcing property for $PLC_1$ one may think that the enforcement of $e_2$ in $PLC_2$ is now superfluous to prevent water overflow in $T_2$. However, this is not the case if the attacker can compromise $PLC_2$. Consider a *second attack* to $PLC_2$, an *integrity attack* that adds an offset of $-30$ to the measured water level of $T_2$. We show a ladder logic implementation of such attack in the right-hand side of Figure 8.3 where, for simplicity, we omit the initial silent silent phases lasting 500 seconds. The impact on the tanks $T_1$ and $T_2$ of the malware injected in $PLC_2$ in the presence of the enforcing of the properties $e_1'$ and $e_3$, respectively, is represented on the upper graphs of Figure 8.6. Again, the red region shows when the attack becomes active. As the reader may notice, the compromised $PLC_2$ never sends requests to close the valve causing the

overflow of the water tank $T_2$. On the other hand, when enforcing the three properties $e'_1, e_2, e_3$ in the three PLCs, the lower graphs of Figure 8.6 shows that the overflow of tank $T_2$ is prevented. Again, the green highlighted regions denote when the monitor *detects* the attack and *mitigates* the commands of the compromised $PLC_2$. Here, the monitor *inserts* the request to close the valve on behalf of $PLC_2$ when $T_2$ reaches a high level.

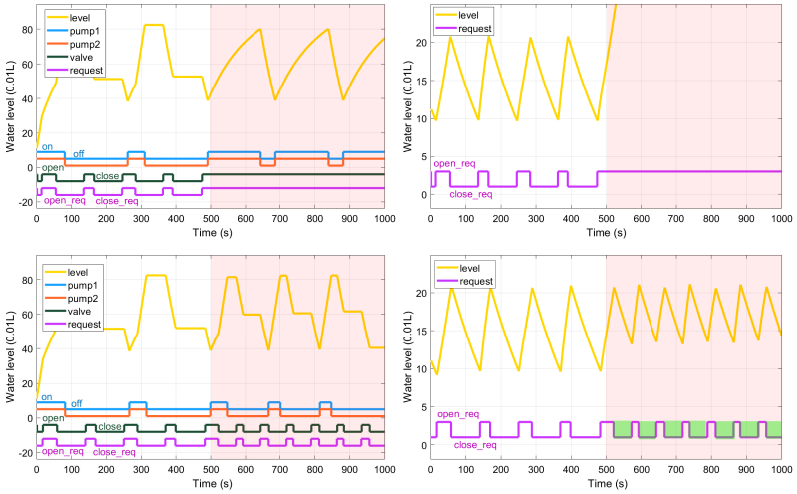



**Figure 8.6:** Tank overflow: integrity attack on $PLC_2$ when enforcing $e'_1, e_3$ (up) and $e'_1, e_2, e_3$ (down).

**Valve damage** We now consider attacks whose goal is to damage the valve via *chattering*, *i.e.*, rapid alternation of openings and closings of the valve that may cause mechanical failures on the long run. In the left-hand side of Figure 8.7 we show a possible ladder logic implementation of a *third attack* that does *injection* of the commands to open and close the valve. In particular, the attack repeatedly alternates a *stand-by phase*, lasting 70 seconds, and a *injection phase*, lasting 30 seconds (yellow region); then, in the injection phase the valve is opened and closed rapidly (red region). With no enforcement, the impact of the attack on the tanks $T_1$ and $T_2$ is represented on the upper graphs

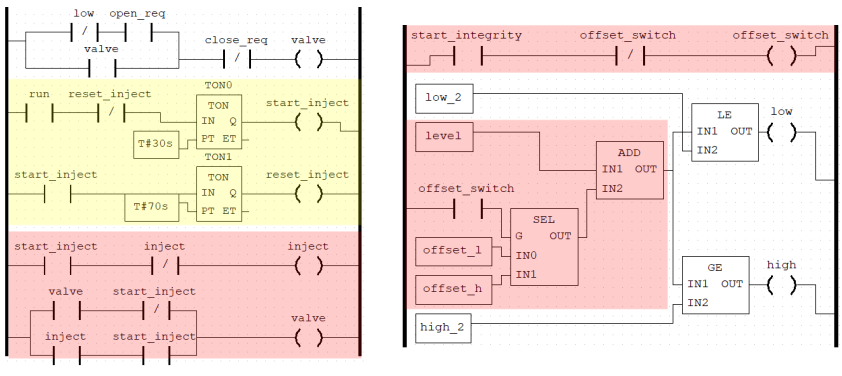



**Figure 8.7:** Valve damage: Ladder logic of the first (left) and the second attack (right).

of Figure 8.8, where the red region denotes when the attack becomes active. From the graph associated to the execution of $T_1$ the reader can easily see that the valve is chattering. Note that this is a *stealthy attack* as the water level of $T_2$ is maintained within the normal operation bounds.

In order to prevent this kind of attacks, we might consider to enforce in $PLC_1$ a bounded mutual exclusion property of the form $e_1'' \triangleq (\mathrm{BME}_{100}\{\overline{\mathsf{open}}, \overline{\mathsf{close}}\})^*$ to ensure that within 100 consecutive scan cycles (10 seconds) openings and the closings of the valve may only occur in mutual exclusion. When the property $e_1''$ is enforced in $PLC_1$, the lower graphs of Figure 8.8 shows that the chattering of the valve is prevented. In particular, the green highlighted regions denote when the monitor *detects* the attack and *mitigates* the commands on the valves of the compromised $PLC_1$.

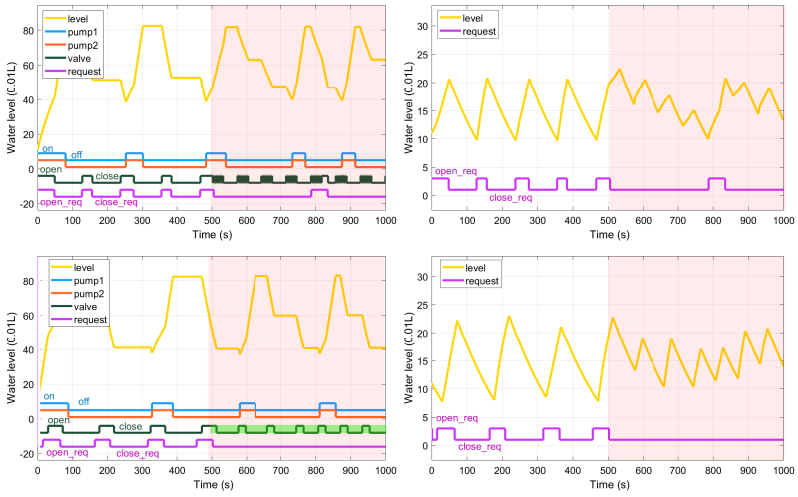



**Figure 8.8:** Valve damage: injection attack on $PLC_1$ in the absence (up) and in the presence (down) of enforcement.

A *fourth attack* with the same goal of chattering the valve may be launched on $PLC_2$, by sending rapidly alternating requests to open and close the valve. This can be achieved by means of an *integrity attack* on the sensor of the tank $T_2$ by rapidly switching the measurements between low and high. In the right-hand side of Figure 8.7 we show parts of the ladder logic implementation of this attack on $PLC_2$, where, for simplicity, we omit the machinery for dealing with the alternation of phases. Again, the attack repeatedly alternates between a *stand-by phase*, lasting 70 seconds, and a *active phase*, lasting 30 seconds. When the attack is in the active phase (red region) the measured water level of $T_2$ rapidly switches between low and high, thus, sending requests to $PLC_1$ to rapidly open and close the valve in alternation.

The impact of this attack targeting on $PLC_2$ in the absence of an enforcing monitor is represented in the upper graphs of Figure 8.9, where the red region shows when the attack becomes active. Notice that the rapid alternating requests originating from $PLC_2$ cause a chattering of the valve. On the other hand, with the enforcement of the property $e_1''$ in $PLC_1$ , the lower graph of Figure 8.9 shows that the correct behaviour of

tanks $T_1$ and $T_2$ is ensured. In that figure, the green highlighted regions denote when the enforcer of $PLC_1$ *detects* the attack and *mitigates* the commands (on the valve) of the compromised $PLC_2$. Notice that in this case no enforcement is required in $PLC_2$.

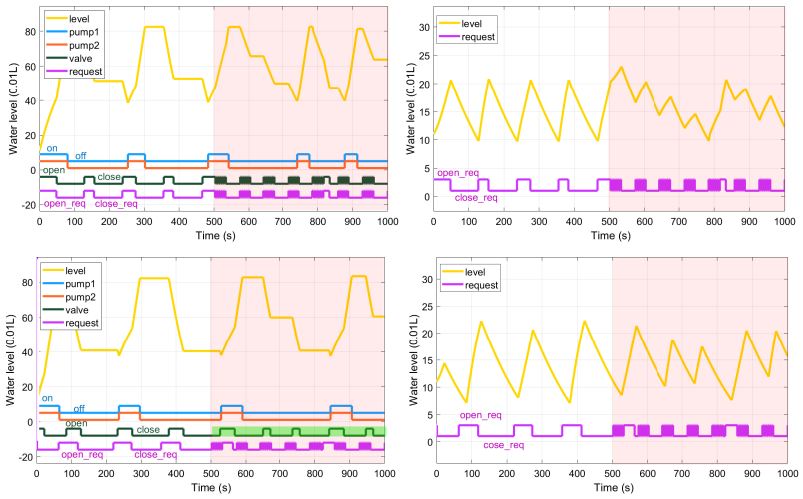



**Figure 8.9:** Valve damage: integrity attack on $PLC_2$ in the absence (up) and in the presence (down) of enforcement.

**Pump damage** Finally, we consider attacks whose goal is the damage of the pumps, and in particular $pump_3$. In that case, an attacker may force the pump to start when the water tank $T_3$ is empty. This can be done with a *fifth attack* that *injects* commands to turn on the pump based on a ladder logic implementation similar to that seen in Figure 8.3. The impact of this attack to tank $T_3$ in the absence of enforcement is represented on the left-hand side graphs of Figure 8.10, where the red region shows when the attack becomes active. As the reader may notice, $pump_3$ is turned on when $T_3$ is empty.

Now, we can prevent damage on $pump_3$ by enforcing on $PLC_3$ the following conditional bounded persistent property: $e'_3 \triangleq (\text{CBP}_{[1,w]}(l_3, \overline{\text{off}_3}))^*$. The enforcement of this property ensures that $pump_3$ is off for $w$ consecutive scan cycles when the level of water in tank $T_3$ is low, for $w < z$ and $z \in \mathbb{N}$ being the time (expressed in scan cycles) required fill up tank $T_3$ when the pump is off. Thus, when the enforcement of the $e'_3$ is active, the lower graphs of Figure 8.10 shows that the correct behaviour of $T_3$ is ensured, thus

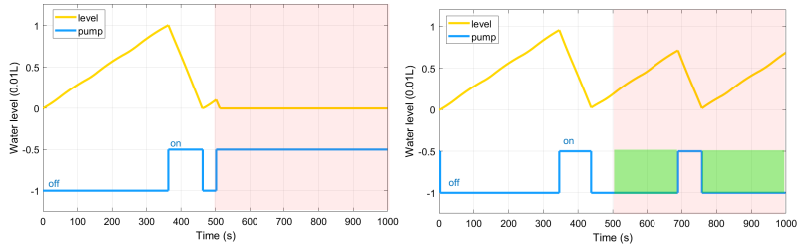



**Figure 8.10:** Pump damage: injection attack on $PLC_3$ in the absence (up) and in the presence (down) of enforcement.

preventing pump damage. In that figure, the green highlighted regions denote when the monitor *detects* the attack and *mitigates* the commands (of the pumps) of the compromised $PLC_3$. More precisely, the enforcer *suppresses* the commands to turn on the pump when the tank is empty, for $w$ consecutive scan cycles.

## 8.4 Summary

In this chapter we have shown the effectiveness of our enforcement via a full implementation of a non-trivial case study with monitors running on FPGAs and enforcing PLCs, running on Raspberry Pi devices. We have relied on Simulink to simulate the physical part of our case study, Figure 8.1. Concerning the computational overhead introduced by the enforcers running on FPGAs, we have shown that the duration of the scan cycle of the monitored PLCs has increased only by a factor of 1%. Furthermore, we have tested our implementation when the enforced PLCs were injected with five different malware aiming at causing three different physical perturbations: (i) overflow the water tanks, (ii) damage of the valve, (iii) damage of the pumps. Recall that in order to simulate the injection of malware in the PLCs, we have reinstalled the original PLC ladder logics with compromised ones, containing some additional logic intended to disrupt the normal operations of the PLC, see for instance Figure 8.3. Finally, we have shown how the enforcement of specific properties have mitigated the attacks by preserving the correct behaviour of the monitored PLCs.

# Chapter 9

# End of Part II

In the second part of this work we have provided a formal approach based on runtime enforcement for the security of control systems. In Chapter 7 we have defined a formal language to express networks of monitored controllers, potentially compromised with colluding malware that may forge/drop actuator commands, modify sensor readings, and forge/drop inter-controller communications. The enforcing monitors have been expressed via a finite-state sub-class of Ligatti's edit automata.

Then, we have defined a simple description language to express ad-hoc *timed regular properties*, always terminating with an end event, which have been used to describe a wide family of correctness properties can be combined in a modular fashion to prescribe precise controller behaviours. For instance, our description language allows us to capture most controller properties collected in [58].

Once defined a formal language to describe controller properties, we have provided a *synthesis function* $\langle\!| - |\!\rangle$ that, given an alphabet $\mathcal{P}$ of observable actions (sensor readings, actuator commands, and inter-controller communications) and a deterministic regular property $e$ consistent with $\mathcal{P}$, returns a finite-state edit automaton $\langle\!| e |\!\rangle^{\mathcal{P}}$. The resulting enforcement mechanism will ensure the required features advocated in the Introduction: transparency, soundness, deadlock-freedom, divergence-freedom, mitigation and compositionality. In particular, with regards to mitigation our synthesised enforcers never suppress end-actions as they are crucial watchdogs signalling the end of a controller scan cycle: when a controller is aligned with its enforcer then a new scan cycle is free to start, otherwise, if this is not the case, the enforcer launches a mitigation cycle by yielding some correct trace, without any involvement of the controller, to reach the completion of the current scan cycle.

In Chapter 8 we have provided a full implementation of a non-trivial case study in the context of industrial water treatment, where enforcers are implemented on FPGAs. In this setting, we showed the effectiveness our enforcement mechanism by means of five carefully-designed attacks targeting the PLCs of our use case. We recall that in the Introduction we have widely discussed about the advantages of securing the enforcing proxy rather than the controller itself.

Finally, notice that malicious alterations of sensor signals at network level, or within the sensor devices, are out of the scope of this work. On the other hand, our ICS architecture ensures that the sensor measurements transmitted to the *supervisory control network* (*e.g.*, to SCADA devices) will not be corrupted by the controller.

## 9.1 Related work

The notion of *runtime enforcement* was introduced by Schneider [135] to enforce security policies via *truncation automata*, a kind of automata that terminates the monitored system in case of violation of the property. Thus, truncation automata can only enforce safety properties.

Ligatti et al. [102] extended Schneider's work by proposing the notion of *edit automata*, *i.e.*, an enforcement mechanism able of *replacing*, *suppressing* and *inserting* system actions. Edit automata are capable of enforcing instances of safety and liveness properties, along with other properties such as renewal properties [22, 102]. In general, Ligatti et al.'s edit automata have an enumerable number of states, whereas in this work we restrict ourselves to finite-state edit automata equipped with Martinelli and Matteucci's operational semantics [108].

Bielova and Massacci [22, 23] provided a stronger notion of enforceability by introducing a *predictability* criterion to prevent monitors from transforming invalid executions in an arbitrary manner. Intuitively, a monitor is said predictable if one can predict the number of transformations used to correct invalid executions, thereby avoiding unnecessary transformations.

Falcone et al. [49, 51] proposed a synthesis algorithm, relying on *Streett automata*, to translate most of the property classes defined within the *safety-progress hierarchy* [107] into enforcers. In the safety-progress hierarchy our global properties can be seen as *guarantee properties* for which all execution traces that satisfy a property contain at least one prefix that still satisfies the property. In the Safety-Progress classification our global properties can be seen as *guarantee properties* for which all execution traces that satisfy a property contain at least one prefix that still satisfies the property.

Beauquier et al. [18] proved that finite-state edit automata (*i.e.* those edit automata we are actually interested in) can only enforce a sub-class of regular properties. Actually they can enforce *all and only* the regular properties that can be recognised by a finite automata whose cycles always contain at least one final state. This is the case of our enforced regular properties, as well-formed local properties in $\mathbb{P}\mathrm{rop}\mathbb{L}$ always terminate with the "final" atomic property $\mathsf{end}$.

Some interesting results on runtime enforcement of *reactive systems* (which have many aspects in common with control systems) have been presented by Könighofer et al. [84]. They defined a synthesis algorithm that given a safety property returns a monitor, called *shield*, that analyses both inputs and outputs of a reactive system, and

enforces the desired property by correcting the minimum number of output actions. More recently, Pinisetty et al. [126, 124] proposed a bi-directional runtime enforcement mechanism for reactive systems, and more generally for cyber-physical systems, to correct both inputs and outputs. They express the desired properties in terms of *Discrete Timed Automata* (DTA) whose labels are system actions. Thus, an execution trace satisfies a required property only if it ends up on a final state of the corresponding DTA. Although the authors do not identify specific classes of correctness properties as we aim to do, DTAs are obviously more expressive than our class of regular properties. However, as not all regular properties can be enforced [18], they proposed a more permissive enforcement mechanism that accepts also execution traces which may reach a final state.

Aceto et al. [5, 4] developed an operational framework to enforce properties in HML logic with recursion ($\mu$HML) relying on suppression. More precisely, they achieved the enforcement of a safety fragment of $\mu$HML by providing a linear automated synthesis algorithm that generates correct suppression monitors from formulas. Here, in order to avoid deadlocks, when the SuS reaches a state in which the monitor does not specify any transformation, the monitor ceases its activity and acts as the go monitor (see also [52]). Enforceability of modal $\mu$-calculus (a reformulation of $\mu$HML) was previously tackled by Martinelli and Matteucci [108] by means of a synthesis algorithm which is exponential in the length of the enforceable formula. More recently, Cassar [32] defined a general framework to compare different enforcement models and different correctness criteria, including optimality. His works focuses on the enforcement of a safety fragment of $\mu$HML, paying attention to both directional and bi-directional notions of enforcement.

It is worth mentioning here runtime adaptation [33, 34], a monitoring technique that sits between runtime verification and runtime enforcement. Violation detections are replaced by adaptation actions that respond to behaviours detected, while reusing as many elements as possible from the system under scrutiny.

Concerning the notion of deterministic enforcement, we synthesized deterministic enforcers as defined in [5, 32], which were derived from syntactically deterministic regular expressions. An quite different notion of deterministic monitoring was introduced in [54, 55], there the authors proposed a contextual definition for deterministic monitoring based on consistent detections, they considered detections (i.e., the verdicts of the monitors) as the only externally visible aspect of a monitor. Thus, given a trace exhibited by the program it is instrumented with, a consistent monitor is required to always reach the same verdict for that trace. Note that, a consistent detection allows such a monitor to pass through different intermediate states during the course of its verdict-reaching trace analysis. Finally, they proposed the notion of controllability that leverages a tractable method for assessing deterministic monitor behaviour.

Concerning syntactically deterministic monitors, i.e., monitors that cannot contain a nondeterministic choice between two sub-monitors reached by the same action, in

[6] the authors showed that for every monitor derived from and HML formula with recursion three is an equivalent, deterministic one, which is at most doubly exponential in size with respect to the original monitor. When monitors are described as CCS-like processes, this doubly exponential bound is optimal. When (deterministic) monitors are described as finite automata, then they can be exponentially more succinct than their CCS process form.

Concerning the features of soundness and transparency, we have relied on the ones defined by Ligatti et al. [102]. Other definitions have been provided. For instance, in [5] the authors enforced properties of the $\mu$HML logic which is a branching-time logic, i.e., the semantics of such properties is given in terms of computation graphs, not traces as in our case. Concerning soundness, it is required that the resulting composite system obtained from instrumenting the enforcer with the system/controller should satisfy the property of interest, whenever this property is satisfiable. On the other hand, transparency requires that whenever a system (controller in our case) already satisfies the property to be enforced, the behaviour of the enforced system should be behaviourally equivalent (in terms of weak bisimilarity) to the original system.

In [126] soundness requires that for any input word the output of the enforcer can be extended to a sequence that satisfies, very much like in Ligatti et al. [102]. On the other hand, transparency requires that any new input event will be simply forwarded by the enforcer if what has been computed as output earlier by the enforcer followed by the input can be extended to a sequence that satisfies the enforced property in the future. This is a somewhat stronger requirement, indeed a transparency requirement according to. [102] is satisfied if the transparency requirement of [126] is satisfied. Similar soundness and transparency definitions, although in timed systems context, have been presented by [127]. There, soundness requires that if a timed word is released as output by the enforcement function, in the future, the output of the enforcement function should satisfy the property. On the other hand, transparency requires that, at any time, the output is a delayed prefix of the observed input.

As regards papers in the context of *control system security* closer to our objectives, McLaughlin [111] proposed the introduction of an enforcement mechanism, called $C^2$, similar to our secure proxy, to mediate the control signals $u_k$ transmitted by the PLC to the plant. Thus, like our secured proxy, $C^2$ is able to suppress commands, but unlike our proxy, it cannot autonomously send commands to the physical devices in the absence of a timely correct action from the PLC. Furthermore, $C^2$ does not seem to cope with inter-controller communications, and hence with colluding malware operating on PLCs of the same field network.

Mohan et al. [114] proposed a different approach by defining an ad-hoc security architecture, called *Secure System Simplex Architecture* (S3A), with the intention to generalise the notion of "correct system state" to include not just the physical state of the plant but also the *cyber state* of the PLCs of the system. In S3A, every PLC runs under the scrutiny of a *side-channel monitor* which looks for deviations with respect to *safe*

*executions*, taking care of real-time constraints, memory usage, and communication patterns. If the information obtained via the monitor differs from the expected model(s) of the PLC, a *decision module* is informed to decide whether to pass the control from the "potentially compromised" PLC to a *safety controller* to maintain the plant within the required safety margins. As reported by the same authors, S3A has a number of limitations comprising: (i) the possible compromising of the side channels used for monitoring, (ii) the tuning of the timing parameters of the state machine, which is still a manual process.

Finally, our work may remind the reader of *supervisory control theory* [130, 27], a general theory for automatic synthesis of controllers (supervisors) for *discrete event systems*, given a plant model and a specification for the controlled behaviour. Fabian and Hellgren [48] have pointed out a number of issues to be addressed when adopting supervisory control theory in industrial PLC-based facilities, such as causality, incorrect synchronisation, and choice between alternative paths. However, in our work we focus on a quite different kind of synthesis: from a correctness specification to an enforcing monitor. As our synthesis regards only logical devices (no plant involved) we are not affected from problems similar to those mentioned by Fabian and Hellgren.

## 9.2 Future work

We are currently working on a user-friendly tool that synthesizes our enforcers from the family of properties that we have presented in Chapter 7. In that tool our properties are represented with a tree-like structure and can be combined with intuitive drag-and-drop actions.

We also want to test our enforcement mechanism in other control system domains, such as robotics and power transmission. More generally we would like to consider physical plants with significant uncertainties, that is, measurement noises and physical process uncertainties. In particular, a significant measurement noise might falsely indicate that the monitored plant is outside safe limits and thus induce our enforcers to take erroneous correcting actions. To address such challenges we could rely on and integrate our enforcers with well-know algorithms from the control theory field. In practice, these algorithms provide the means to correctly estimate the state of the physical plant even when it is affected by significant uncertainties.

Finally, we would like to enhance the capabilities of our enforcers to deal with malicious alterations of sensor measurement coming from compromised sensor devices. To do so we will investigate the integration of our secured proxies with *physics-based attack detection* [61]. Since the differential/difference equations are involved in the calculation of such algorithms, these could introduce significant overhead into the secure proxy scan cycle and therefore into the monitored controller scan cycle.

# Chapter 10

# Overview of published work

We now overview the research papers that the author of this thesis has contributed to during the course of his PhD.

Concerning the first part of this work, we made our first steps in the static analysis for the security of ICSs via model checking in the 38-th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE). The paper is cited below and contains the results and the contributions presented in Chapter 4.

- R. Lanotte, M. Merro, and A. Munteanu. 2018. A Modest Security Analysis of Cyber-Physical Systems: A Case Study. In FORTE (LNCS), Vol. 10854. Springer, 58–78.

The author implemented and analysed the case study under the guidance of the co-authors i.e., his PhD advisers. In addition, he was also extensively involved in writing up the paper.

As regards the approach via statistical model checking, in Chapter 5 we mentioned the work published in the journal ACM Transactions on Privacy and Security (TOPS).

- R. Lanotte, M. Merro, A. Munteanu, and L. Viganò. A Formal Approach to Physics-based Attacks in Cyber-physical Systems. ACM Trans. Priv. Secur. 23, 1 (2020), 3:1–3:41.

The author's main contribution lies in the implementation section where he implemented and analysed the case study of the paper.

In Chapter 5, we also have described in a detailed manner the work published in the 8-th International Conference on Formal Methods in Software Engineering (FormaliSE'20).

- A. Munteanu, M. Pasqua, M. Merro. Impact Analysis of Cyber-Physical Attacks on a Water Tank System via Statistical Model Checking. In IEEE/ACM 8th Int. Conf. on Formal Methods in Sw Eng. (FormaliSE), pp. 34-43, 2020.

There the author was extensively involved in writing up the paper. In addition, the author contributed again with the implementation and the analysis of the case study.

As regards the second part of this thesis, i.e., runtime enforcement for the security of ICSs. Chapters 7 and 8 contain the work that have been submitted for reviews to a journal.

- R. Lanotte, M. Merro, and A. Munteanu. Runtime Enforcement of Programmable Logic Controllers. Submitted to journal.

Once again, the author was involved extensively in writing up the paper. In addition, he formulated the formal models and proofs presented in the paper under guidance of the PhD advisers. Furthermore, he also contributed with the implementation part.

In the works mentioned below, the author was involved extensively in writing up the paper. In addition, he formulated the formal models and proofs presented in the paper under guidance of the PhD advisers. Chapters 7 and 8 contain an extended version of the paper appeared in the 33-rd IEEE Computer Security Foundations Symposium (CSF 2020).

- R. Lanotte, M. Merro, and A. Munteanu. 2020. Runtime Enforcement for Control System Security. In CSF. IEEE, 246–261.

This work is based on our first steps in runtime enforcement published in the 21-st Italian Conference on Theoretical Computer Science (ICTCS 2020).

- R. Lanotte, M. Merro, and A. Munteanu. 2020. A process calculus approach to correctness enforcement of PLCs. In ICTCS (CEUR Workshop Proceedings, Vol. 2756). CEUR-WS.org, 81–94.

This work has been revised and extended with our preliminary steps in the implementation of our enforcement mechanism which will appear in the journal Theoretical Computer Science (TCS).

- R. Lanotte, M. Merro, and A. Munteanu. A process calculus approach to detection and mitigation of PLC malware. To appear in the journal Theoretical Computer Science.

In this work the author also contributed with the implementation part.

# Chapter 11

# Appendix

## 11.1 An introduction to the HMODEST language

A HMODEST specification consists of a sequence of declarations (constants, variables, actions, and sub-processes) and a main process behaviour. The most simple process behaviour is expressed by (prefixing) *actions* that may be used to synchronize parallel components. The construct `do` models loops, *i.e.*, unguarded iterations that can be exited via the special action `break`. There is a construct `par` to launch two or more processes in *parallel*, according to an interleaving semantics. The construct `alt` models *nondeterministic choices*. The `invariant` construct is used to control the evolution of continuous variables. Furthermore, all constructs can be decorated with guards, to represent enabling conditions, by means of the `when` construct. We can use both `invariant` and `when` constructs to specify that a behaviour should be executed after a precise amount of time. Thus, we can write `invariant`$(\texttt{c} \leq \texttt{k})$ `when`$(\texttt{c} \geq \texttt{k})$ *P()*, where `c` is a clock variable and `k` a real value, to model that the process *P()* may start its execution only after `k` instants; if $\texttt{k} = 0$ then the execution of *P()* may start immediately.

In order to explain these constructs, we model a small example. Consider a *Master* process and a *Slave* process that may synchronize via the actions *go* and *end* to allow the *Master* to send instructions to the *Slave*. Depending on the received instructions, the *Slave* either restarts or it sleeps for one time unit and then ends its execution. Once synchronized via the *go* action, the *Master* sleeps for two time units and then synchronizes on the *end* action. More precisely,

*Master* :

```
synchronize on action go
sleep 2 time units
synchronize on action end
```

*Slave* :

```
repeat
  listen for action synchronization
  if action is go then
    continue
  else
    sleep 1 time unit
    exit
```

```
// action declarations
action  go, end;
// process declarations
process Master() {
    clock cm; // local clock declaration
    invariant(cm <= 0) when(cm >= 0) go {= cm = 0 =};
    invariant(cm <= 2) when(cm >= 2) end
}
process Slave() {
    clock cs; // local clock declaration
    do {
        alt {
            :: go
            :: end {= cs = 0 =}; invariant(cs <= 1) when(cs >= 1) break
        }
    }
}
// main
par { :: Master() :: Slave() }
```

**Figure 11.1:** Master and Slave processes in HMODEST

The compound system is given by running the two processes *Master* and *Slave* concurrently.

Figure 11.1 shows an implementation in HMODEST of the system above. Both *Master* and *Slave* processes declare private clocks that are reset each time is necessary to impose a specific time delay. The communication is implemented via the two actions *go* and *end*; the testing is via pattern matching within nondeterministic choice.

## 11.2 Proofs of Section 7.4

In order to prove the results of Section 7.4, in Table 11.1 we provide a formal definition of cross product between two edit automata. The first three cases are straightforward, we explain only the fourth case, thus, we consider the cross product associated to $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\sum_{i\in I}\lambda_i.\mathsf{E}_i, \sum_{j\in J}\nu_j.\mathsf{E}_j)$. In this case, the result is either a suppression edit automaton, or an edit automaton that: (i) *allows* all common actions $\lambda_i$, for $\lambda_i = \nu_j$, allowed by $\mathsf{E}_i$ and $\mathsf{E}_j$, (ii) *inserts* such common actions $\lambda_i$ whenever the controller is about to complete the scan cycle, *i.e.*, it is about to emit an action $\mathsf{end}$, and (iii) *suppresses* all others actions. Here, the index set is $H = \{(i,j) \in I \times J : \lambda_i = \nu_j \in \mathcal{P} \text{ and } \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\mathsf{E}_i, \mathsf{E}_j) \neq \sum_{\alpha\in\mathcal{P}\setminus\{\mathsf{tick},\mathsf{end}\}} {}^{-}\alpha.\mathsf{X}\}$. Intuitively, $H$ is the set of pairs $(i,j)$ of indexes in $I \times J$ for which the unfolding of the product of the associated edit automata $\mathsf{E}_i$ and $\mathsf{E}_j$ never ends into a suppression-only edit automata.

Let us prove the complexity of the synthesis algorithm formalised in Proposition 1. For that we need a couple of technical lemmata. The first lemma extends classical results on the complexity of cross product of finite state automata to cross product of edit automata.

**Lemma 1** (Complexity of cross product)**.** *Let* $\mathsf{E}_1, \mathsf{E}_2 \in \mathbb{E}\mathtt{dit}$ *be two edit automata and* $\mathcal{P}$ *be a set of observable actions of size* $n$*. Let* $v_1, v_2$ *be the number of derivatives of* $\mathsf{E}_1$ *and*

$$
\begin{array}{rcl}
\mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\mathsf{X}_1,\mathsf{X}_2) & \triangleq & \mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\mathsf{E}_1,\mathsf{E}_2),\ \text{if } \mathsf{X}_1=\mathsf{E}_1 \text{ and } \mathsf{X}_2=\mathsf{E}_2 \\
\mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\mathsf{X},\sum_{i\in I}\lambda_i.\mathsf{E}_i) & \triangleq & \mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\mathsf{E},\sum_{i\in I}\lambda_i.\mathsf{E}_i),\ \text{if } \mathsf{X}=\mathsf{E} \\
\mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\sum_{i\in I}\lambda_i.\mathsf{E}_i,\mathsf{X}) & \triangleq & \mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\sum_{i\in I}\lambda_i.\mathsf{E}_i,\mathsf{E}),\ \text{if } \mathsf{X}=\mathsf{E} \\
\mathsf{Prod}^{\mathcal{P}}_{\mathsf{Z}}(\sum_{i\in I}\lambda_i.\mathsf{E}_i,\sum_{j\in J}\nu_j.\mathsf{E}_j) & \triangleq & \begin{cases} \sum_{\alpha\in\mathcal{P}\setminus\{\mathsf{tick},\mathsf{end}\}} {}^{-}\alpha.\mathsf{Z} & \text{if } H=\emptyset \\ \sum_{(i,j)\in H}(\lambda_i.\mathsf{X}_{i,j}+\lambda_i\prec\mathsf{end}.\mathsf{X}_{i,j})+\sum_{\alpha\in(\mathcal{P}\setminus\{\mathsf{tick},\mathsf{end}\})\setminus\bigcup_{(i,j)\in H}\lambda_i} {}^{-}\alpha.\mathsf{Z} & \text{if } H\neq\emptyset \\ \text{for } \mathsf{X}_{i,j}=\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i,\mathsf{E}_j) & \end{cases}
\end{array}
$$

**Table 11.1:** Cross product between two edit automata with alphabet $\mathcal{P}$.

$\mathsf{E}_2$, *respectively.*[1] *Let* $w_1, w_2$ *be the number of distinct observable actions that can be fired by* $\mathsf{E}_1$ *and* $\mathsf{E}_2$ *and their derivatives, respectively. The complexity of the algorithm to compute* $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\mathsf{E}_1,\mathsf{E}_2)$ *is* $\mathcal{O}(v_1\cdot v_2\cdot(\max(w_1,w_2)+n))$.

The second lemma provides an upper bound to the number of derivates of the automaton $\langle\!\|\ e\ \|\!\rangle^{\mathcal{P}}$. For that we need a formal definition of size of both global and local properties. Intuitively, the size of a property is given by the number of operators occurring in it.

**Definition 8.** *Let* $\mathsf{dim}() : \mathbb{P}\mathrm{rop}\mathbb{G}\cup\mathbb{P}\mathrm{rop}\mathbb{L}\to\mathbb{N}$ *be a property-size function defined as:*

$$
\begin{array}{rclrcl}
\mathsf{dim}(p^*) & \triangleq & \mathsf{dim}(p) & \mathsf{dim}(e_1\cap e_2) & \triangleq & \mathsf{dim}(e_1)+\mathsf{dim}(e_2)+1 \\
\mathsf{dim}(\epsilon) & \triangleq & 1 & \mathsf{dim}(p_1;p_2) & \triangleq & \mathsf{dim}(p_1)+\mathsf{dim}(p_2)+1 \\
\mathsf{dim}(p_1\cap p_2) & \triangleq & \mathsf{dim}(p_1)+\mathsf{dim}(p_2)+1 & \mathsf{dim}(\bigcup_{i\in I}\alpha_i.p_i) & \triangleq & \mid I\mid+\sum_{i\in I}\mathsf{dim}(p_i).
\end{array}
$$

**Lemma 2** (Upper bound of number of derivatives)**.** *Let* $e\in\mathbb{P}\mathrm{rop}\mathbb{G}$ *be a global property with and* $m=\mathsf{dim}(e)$*, and* $\mathcal{P}$ *be a set of observable actions. Then, the number of derivatives of* $\langle\!\|\ e\ \|\!\rangle^{\mathcal{P}}$ *is at most* $m^{k+1}$*, where* $k$ *us the number of occurrences of the symbol* $\cap$ *in* $e$*.*

*Proof.* The proof is by structural induction on $e$.

Let $e\equiv e_1\cap e_2$ and $m=\mathsf{dim}(e_1\cap e_2)$. By definition, the synthesis function recalls itself on $e_1$ and $e_2$. Obviously, $m_1+m_2=m-1$ with $m_1=\mathsf{dim}(e_1)$ and $m_2=\mathsf{dim}(e_2)$. Let $k$, $k_1$ and $k_2$ be the number of occurrences of the symbol $\cap$ in $e_1\cap e_2$, $e_1$ and $e_2$, respectively. We deduce that $k_1+k_2=k-1$. By inductive hypothesis, $\langle\!\|\ e_1\ \|\!\rangle^{\mathcal{P}}$ has at most $m_1^{k_1+1}$ derivatives, and, $\langle\!\|\ e_2\ \|\!\rangle^{\mathcal{P}}$ has at most $m_2^{k_2+1}$ derivatives. As the synthesis returns the cross product between $\langle\!\|\ e_1\ \|\!\rangle^{\mathcal{P}}$ and $\langle\!\|\ e_2\ \|\!\rangle^{\mathcal{P}}$, we derive that the resulting edit automaton will have at most $m_1^{k_1+1}\cdot m_2^{k_2+1}$ derivatives. The result follows because $m_1^{k_1+1}\cdot m_2^{k_2+1}\leq m^{k_1+1}\cdot m^{k_2+1}\leq m^{k_1+k_2+2}\leq m^{k-1+2}\leq m^{k+1}$.

Let $e\equiv p^*$, for $p\in\mathbb{P}\mathrm{rop}\mathbb{L}$. In order to analyse this case, as $\langle\!|p^*|\!\rangle^{\mathcal{P}}\triangleq\mathsf{X}$, for $\mathsf{X}=\langle\!\|\ p\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, we proceed by structural induction of $p\in\mathbb{P}\mathrm{rop}\mathbb{L}$. We focus on the most significant case $p\equiv\bigcup_{i\in I}\pi_i.p_i$. Let $p\equiv\bigcup_{i\in I}\pi_i.p_i$ and $m=\mathsf{dim}(\bigcup_{i\in I}\pi_i.p_i)$. By definition the synthesis produces $\mid I\mid$ derivatives, one for each $\pi_i\in I$, and also the

[1] These numbers are finite as we deal with finite-state edit automata.

derivative Z. Furthermore, the synthesis algorithm re-calls itself $\mid I \mid$ times on $p_i$, with $m_i = \mathsf{dim}(p_i)$ such that $m = \mid I \mid + \sum_{i \in I} m_i$, for $i \in I$. Let $k$ and $k_i$ be the number of occurrences of $\cap$ in $p$ and in $p_i$, respectively, for $i \in I$. We deduce that $\sum_{i \in I} k_i = k$. By inductive hypothesis, the synthesis produces $m_i^{k_i+1}$ derivatives on each property $p_i$, for $i \in I$. Summarising, in this case the number of derivatives is $1 + \mid I \mid + \sum_{i \in I} m_i^{k_i+1}$. Finally, the thesis follows as $1 + \mid I \mid + \sum_{i \in I} m_i^{k_i+1} \leq \sum_{i \in I} m^{k_i+1} \leq m^{k+1}$. □

*Proof of Proposition 1 (Complexity).* For any property $e \in \mathbb{P}\mathtt{rop}\mathbb{G}$, we prove that the recursive structure of the function returning $\langle\!\langle\, e \,\rangle\!\rangle^{\mathcal{P}}$ can be characterised in the following form: $T(m) = T(m-1) + m^{k+1}$, with $m = \mathsf{dim}(e)$, and $k$ the number of occurrences of $\cap$ in $e$. The result follows because $T(m) = T(m-1) + m^{k+1}$ is $\mathcal{O}(m^{k+1})$. The proof is by case analysis on the structure of $e$, by examining each synthesis step in which the synthesis process $m = \mathsf{dim}(e)$ symbols.

Case $e \equiv e_1 \cap e_2$. Let $m = \mathsf{dim}(e_1 \cap e_2)$. By definition, the synthesis $\langle\!\langle\, e_1 \cap e_2 \,\rangle\!\rangle^{\mathcal{P}}$ call itself on $e_1$ and $e_2$, with $m_1 = \mathsf{dim}(e_1)$ and $m_2 = \mathsf{dim}(e_2)$ symbols, respectively, where $m_1 + m_2 = m - 1$. Let $k$ be the number of occurrences of $\cap$ in $e$ and $k_1, k_2$ be the number of occurrences of $\cap$ in $e_1$ and $e_2$, respectively. We deduce that $k_1 + k_2 = k - 1$. By Lemma 1 and by Lemma 2 we know that number of operations required for the cross product between $\langle\!\langle\, e_1 \,\rangle\!\rangle^{\mathcal{P}}$ and $\langle\!\langle\, e_2 \,\rangle\!\rangle^{\mathcal{P}}$ is $m_1^{k_1+1} \cdot m_2^{k_2+1} \cdot \max(m_1, m_2)$. Thus, we can characterise the recursive structure as: $T(m) = T(m_1) + T(m_2) + m_1^{k_1+1} \cdot m_2^{k_2+1} \cdot \max(m_1, m_2)$. We notice that the complexity of this recursive form is smaller than the complexity of $T(m-1) + m^{k+1}$.

Case $e \equiv p^*$. In order to prove this case, as $\langle\!\langle\, p^* \,\rangle\!\rangle^{\mathcal{P}} \triangleq \mathsf{X}$, for $\mathsf{X} = \langle\!\langle\, p \,\rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, we proceed by case analysis on $p \in \mathbb{P}\mathtt{rop}\mathbb{L}$. Thus, we consider the local properties $p \in \mathbb{P}\mathtt{rop}\mathbb{L}$. We focus on the most significant case $p \equiv \bigcup_{i \in I} \pi_i.p_i$. Let $m = \mathsf{dim}(\bigcup_{i \in I} \pi_i.p_i)$. By definition, the synthesis $\langle\!\langle\, \bigcup_{i \in I} \pi_i.p_i \,\rangle\!\rangle^{\mathcal{P}}$ consumes all events $\pi_i$, for $i \in I$. The synthesis algorithm re-calls itself $\mid I \mid$ times on $p_i$, with $\mathsf{dim}(p_i)$ symbols, for $i \in I$. Furthermore, let $l$ be the size of the set $\mathcal{P}$, the algorithm performs at most $l$ operations due to a summation over over $\alpha \in \mathcal{P} \setminus (\bigcup_{i \in I} \pi_i \cup \{\mathsf{tick}, \mathsf{end}\})$, with $\mid \mathcal{P} \setminus (\bigcup_{i \in I} \pi_i \cup \{\mathsf{tick}, \mathsf{end}\}) \mid < l$. Thus, we can characterise the recursive structure as $T(m) = \sum_{i \in I} T(\mathsf{dim}(p_i)) + l$. Since $\sum_{i \in I} \mathsf{dim}(p_i) = m - \mid I \mid \leq m - 1$. The resulting complexity is smaller than that of $T(m-1) + m^{k+1}$. □

In order to prove Proposition 2 we give a technical results saying that the edit automata synthesised from deterministic global properties are deterministic as well.

**Lemma 3.** *Let $e \in \mathbb{P}\mathtt{rop}\mathbb{G}$ be a deterministic property over a set of observable actions $\mathcal{P}$. Let $\mathsf{E} \xrightarrow{t} \mathsf{E}$ for some trace $t$ and automaton $\mathsf{E}$. If $\mathsf{E} \xrightarrow{\lambda_1} \mathsf{E}_1$ and $\mathsf{E} \xrightarrow{\lambda_2} \mathsf{E}_2$, with a $\mathsf{E}_1 \neq \mathsf{E}_2$, then $\lambda_1 \neq \lambda_2$.*

*Proof.* A property $e \in \mathbb{P}\mathtt{rop}\mathbb{G}$ is deterministic if any sub-term $\cup_{i \in I} \pi_i.p_i$ is such that $\pi_k \neq \pi_h$, for any $k, h \in I$, $k \neq h$. By construction, the cross product of edit automata

preserves determinism. The result follows by reasoning on structural induction on the property $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$. □

*Proof of Proposition 2 (Semantically-deterministic-enforcement).* We show that for any execution trace $t$ and actions $\alpha_1, \alpha_2$ such that $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{t} \mathsf{E} \bowtie J$ and $\mathsf{E} \bowtie J \xrightarrow{\alpha_1} \mathsf{E}_1 \bowtie J_1$ and $\mathsf{E} \bowtie J \xrightarrow{\alpha_2} \mathsf{E}_2 \bowtie J_2$ and $\mathsf{E}_1 \neq \mathsf{E}_2$, it holds that $\alpha_1 \neq \alpha_2$. We proceed by contradiction. Assume there is an $\alpha$ such that $\mathsf{E} \bowtie J \xrightarrow{\alpha} \mathsf{E}_1 \bowtie J_1$ and $\mathsf{E} \bowtie J \xrightarrow{\alpha} \mathsf{E}_2 \bowtie J_2$ and $\mathsf{E}_1 \neq \mathsf{E}_2$. The first transition can be derived either via rule (Allow) or via rule (Insert). We focus on the rule (Allow) (the other case is similar). Suppose $\alpha \neq \mathsf{end}$. Since $\mathsf{E} \bowtie J \xrightarrow{\alpha} \mathsf{E}_1 \bowtie J_1$ is derived via rule (Allow), then $\mathsf{E} \xrightarrow{\alpha} \mathsf{E}_1$ and $J \xrightarrow{\alpha} J_1$. From $J \xrightarrow{\alpha} J_1$ and $\alpha \neq \mathsf{end}$ we derive that $J \xrightarrow{\mathsf{end}}$ and so $\mathsf{E} \bowtie J \xrightarrow{\alpha} \mathsf{E}_2 \bowtie J_2$ is derived via rule (Allow) as well, and $\mathsf{E} \xrightarrow{\alpha} \mathsf{E}_2$. By Lemma 3 it follows the contradiction $\mathsf{E}_1 = \mathsf{E}_2$.. The case $\alpha = \mathsf{end}$ can be treated in a similar manner. □

The next step is the proof of transparency, *i.e.*, the proof of Theorem 1.

We start proving that the cross product between edit automata satisfies a standard correctness result saying that any execution trace associated to the intersection of two regular properties is also a trace of the the cross product of the edit automata associated to the two properties, and vice versa. In order to prove that, given a prefix $\lambda \in \{\alpha, \alpha \prec \beta, {}^{-}\alpha\}$, we write $enfAct(\lambda)$ to denote the resulting action of the monitored controller. Formally, $enfAct(\alpha) = enfAct(\alpha \prec \beta) = \alpha$ and $enfAct({}^{-}\alpha) = \tau$. This notation can be easily extended to a trace $t = \lambda_1 \cdots \lambda_n$ by defining $enfAct(t) = enfAct(\lambda_1) \cdots enfAct(\lambda_n)$.

**Lemma 4** (Correctness of Cross Product)**.** *Let $e_1, e_2 \in \mathbb{P}\mathrm{rop}\mathbb{G}$ (resp., $p_1, p_2 \in \mathbb{P}\mathrm{rop}\mathbb{L}$) and $\mathcal{P}$ be a set of actions such that $\mathsf{events}(e_1 \cap e_2) \subseteq \mathcal{P}$ (resp., $\mathsf{events}(p_1 \cap p_2) \subseteq \mathcal{P}$). Then, it holds that:*

- *If $t$ is a trace of $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, e_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, e_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$ (resp., $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$), then $\widehat{enfAct(t)}$ is prefix of some trace in the semantics $[\![e_1 \cap e_2]\!]$ (resp., $[\![p_1 \cap p_2]\!]$).*
- *If $t$ is a trace in the semantics $[\![e_1 \cap e_2]\!]$ (resp., $[\![p_1 \cap p_2]\!]$) then there exists a trace $t'$ of $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, e_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, e_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$ (resp., $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$) such that $\widehat{enfAct(t')} = t$.*

In the following, we use the symbol $\preccurlyeq$, with $\preccurlyeq \subseteq (\mathbb{P}\mathrm{rop}\mathbb{L} \cup \mathbb{P}\mathrm{rop}\mathbb{G}) \times (\mathbb{P}\mathrm{rop}\mathbb{L} \cup \mathbb{P}\mathrm{rop}\mathbb{G})$, to denote the reflexive and transitive closure of sub-term inclusion between regular expressions such that whenever $p_1 \preccurlyeq p_1'$ and $p_2 \preccurlyeq p_2'$ then $p_1 \cap p_2 \preccurlyeq p_1' \cap p_2'$ and $p_1; p_2' \preccurlyeq p_1'; p_2'$.

*Proof of Theorem 1 (Transparency).* We actually prove a stronger result. Let $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ a global property and $P \in \mathbb{C}\mathrm{tr}\mathbb{l}$ such that $\mathsf{go} \bowtie P \xrightarrow{t} \mathsf{go} \bowtie J$ for some trace $t$. If $t$ is the prefix of some trace in the semantics $[\![e]\!]$ we prove that:

1. $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{t} \mathsf{E} \bowtie J$ in which either $\mathsf{E} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p' \in \mathbb{P}\mathrm{rop}\mathbb{L}$ such that $p' \preccurlyeq e$ and some automaton variable $\mathsf{X}$.
2. There is a trace $t' \in [\![p']\!]$ such that $t \cdot t'$ is a prefix of some trace in $[\![e]\!]$.

These two points imply the result. We proceed by induction on the length $n$ of the execution trace $t$.

*Base case:* $n = 1$. We have that $\mathsf{go} \bowtie P \xrightarrow{\alpha} \mathsf{go} \bowtie J$ with $\alpha \in \mathsf{Sens} \cup \mathsf{Chn}^* \cup \overline{\mathsf{Act}} \cup \{\mathsf{tick}, \mathsf{end}\}$ and $\alpha$ be a prefix of some trace in $[\![e]\!]$. We proceed by induction on the structure of $e$.

*Case* $e \equiv p^*$, for some $p \in \mathbb{P}\mathrm{rop}\mathbb{L}$. We prove by induction on the structure of $p$ the following two results: i) $\langle\!|\ p\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}'} \bowtie P \xrightarrow{\alpha} \mathsf{E} \bowtie J$, where either $\mathsf{E} = \langle\!|\ p'\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}'}$ or $\mathsf{E} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\ p'\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}'}$, for some $p' \in \mathbb{P}\mathrm{rop}\mathbb{L}$ such that $p' \preccurlyeq p$, and some automaton variable $\mathsf{X}'$; ii) there is a trace $t'' \in [\![p']\!]$ such that $\alpha \cdot t''$ is a prefix of some trace in $[\![p]\!]$. As $\langle\!|\ p^*\ |\!\rangle^{\mathcal{P}} \triangleq \mathsf{X}$, with $\mathsf{X} = \langle\!|\ p\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ results i) and ii) imply the required results (1) and (2) for $e = p^*$. We give the cases for $p \equiv p_1; p_2$ and $p \equiv p_1 \cap p_2$; the other case are similar or simpler.

Let $p \equiv p_1; p_2$. In this case, $\alpha$ is a prefix of some trace in $[\![p_1; p_2]\!]$ and $\langle\!|\ p_1; p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ returns $\langle\!|\ p_1\ |\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$, for $\mathsf{Z}' = \langle\!|\ p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, and $\mathsf{Z}' \neq \mathsf{X}$. We prove the two items for the case $p_1 \neq \epsilon$, the case $p_1 = \epsilon$ is simpler.

- Let us prove i). From $p_1 \neq \epsilon$ we derive that $\alpha$ is a prefix of some trace in $[\![p_1]\!]$. From this fact and since $\mathsf{go} \bowtie P \xrightarrow{\alpha} \mathsf{go} \bowtie J$, we derive by inductive hypothesis that $\langle\!|\ p_1\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}} \bowtie P \xrightarrow{\alpha} \mathsf{E}_1 \bowtie J$, where either $\mathsf{E}_1 = \langle\!|\ p_1'\ |\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ or $\mathsf{E}_1 = \mathsf{Z}_1$, with $\mathsf{Z}_1 = \langle\!|\ p_1'\ |\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$, for some $p_1' \in \mathbb{P}\mathrm{rop}\mathbb{L}$ such that $p_1' \preccurlyeq p_1$. Let us analyse $\mathsf{E}_1 = \langle\!|\ p_1'\ |\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ (the analysis for $\mathsf{E}_1 = \mathsf{Z}_1$, with $\mathsf{Z}_1 = \langle\!|\ p_1'\ |\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ is similar). From $\langle\!|\ p_1\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}} \bowtie P \xrightarrow{\alpha} \mathsf{E}_1 \bowtie J$, we derive that $\langle\!|\ p_1; p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}} \bowtie P \xrightarrow{\alpha} \mathsf{E} \bowtie J$, for some $\mathsf{E}$. Moreover, since $\mathsf{E}_1 = \langle\!|\ p_1'\ |\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ with $\mathsf{Z}' = \langle\!|\ p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ and $p_1' \preccurlyeq p_1$, then, by definition of the synthesis algorithm, it follows that $\mathsf{E} = \langle\!|\ p_1'; p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ for $p_1'; p_2 \preccurlyeq p_1; p_2 \equiv p$, as required.
- Let us prove ii). Again, from $p_1 \neq \epsilon$ we derive by inductive hypothesis that there exists $t' \in [\![p_1']\!]$ such that $\alpha \cdot t'$ is a prefix of some trace in $[\![p_1]\!]$. Thus, there is a trace $t''$ such that $t' \cdot t'' \in [\![p_1'; p_2]\!]$ and hence $\alpha \cdot t' \cdot t''$ is a prefix of some trace in $[\![p_1; p_2]\!]$, as required.

Let $p \equiv p_1 \cap p_2$. In this case, we have that $\alpha$ is prefix of some trace in $[\![p_1 \cap p_2]\!]$ and the synthesis of algorithm applied to $\langle\!|\ p_1 \cap p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ returns $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\ p_1\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\ p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}})$.

- Let us prove i). We start by analysing the transitions afforded by the edit automaton $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\ p_1\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\ p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}})$. By definition of cross product in Table 11.1, the most interesting case is when $\langle\!|\ p_1\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \sum_{i \in I} \lambda_i.\mathsf{E}_i$ and $\langle\!|\ p_2\ |\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \sum_{j \in J} \nu_j.\mathsf{E}_j$. In this case, $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\sum_{i \in I} \lambda_i.\mathsf{E}_i, \sum_{j \in J} \nu_j.\mathsf{E}_j)$ is equal to:

$$\begin{cases} \sum\limits_{\alpha \in \mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}} {}^{-}\alpha.\mathsf{X} & \text{if } H = \emptyset \\ \sum\limits_{(i,j) \in H} (\lambda_i.\mathsf{X}_{i,j} + \lambda_i \prec \mathsf{end}.\mathsf{X}_{i,j}) + \sum\limits_{\alpha \in (\mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}) \setminus \bigcup_{(i,j) \in H} \lambda_i} {}^{-}\alpha.\mathsf{X} & \text{if } H \neq \emptyset \\ \text{for } \mathsf{X}_{i,j} = \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) \end{cases}$$

  with $H = \{(i,j) \in I \times J : \lambda_i = \nu_j \in \mathcal{P} \text{ and } \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\mathsf{E}_i, \mathsf{E}_j) \neq \sum_{\alpha \in \mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}} {}^{-}\alpha.\mathsf{X}\}$. The first case is not admissible as we assume that $\alpha$ is a prefix of some trace in

$[\![p_1 \cap p_2]\!]$, thus, the edit automaton associated to $p_1 \cap p_2$ may not be a suppression automaton. Thus, we focus on the second case and the following admissible transitions: $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\lambda_i} \mathsf{X}_{i,j}$, with $\mathsf{X}_{i,j} = \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j)$, for any $(i,j) \in H$. Now, since $\alpha$ is a prefix of some trace in $[\![p_1 \cap p_2]\!]$, then $\alpha$ is a prefix of some trace in both $[\![p_1]\!]$ and $[\![p_2]\!]$. Thus, since $\mathsf{go} \bowtie P \xrightarrow{\alpha} \mathsf{go} \bowtie J$, by inductive hypothesis we have that for $h \in \{1,2\}$, $\langle\!|\, p_h \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} \bowtie P \xrightarrow{\alpha} \mathsf{E}'_h \bowtie J$, where either $\mathsf{E}'_h = \langle\!|\, p'_h \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}'_h = \mathsf{Z}_h$, with $\mathsf{Z}_h = \langle\!|\, p'_h \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p'_h \in \mathbb{P}\mathrm{rop}\mathbb{L}$ such that $p'_h \preccurlyeq p_h$. Hence, by Lemma 4 and by definition of cross product, there exists $(i,j) \in H$ such that $\alpha = \lambda_i$ and $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \bowtie P \xrightarrow{\alpha} \mathsf{E} \bowtie J$ with $\mathsf{E} = \mathsf{X}_{i,j}$, $\mathsf{X}_{i,j} = \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}'_1, \mathsf{E}'_2)$ and $\mathsf{E}'_1 = \mathsf{E}_i$ and $\mathsf{E}'_2 = \mathsf{E}_j$. Thus, $\mathsf{X}_{i,j} = \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j)$. It remains to prove that either $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) = \langle\!|\, p'_1 \cap p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}'}$, or $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p'_1 \cap p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}'}$, such that $p'_1 \cap p'_2 \preccurlyeq p_1 \cap p_2$, for some automaton variable $\mathsf{X}'$. We have proved that for $h \in \{1,2\}$, either $\mathsf{E}'_h = \langle\!|\, p'_h \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}'_h = \mathsf{Z}_h$, with $\mathsf{Z}_h = \langle\!|\, p'_h \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p'_h \in \mathbb{P}\mathrm{rop}\mathbb{L}$ such that $p'_h \preccurlyeq p_h$. Moreover, we have proved also that $\mathsf{E}_i = \mathsf{E}'_1$ and $\mathsf{E}_j = \mathsf{E}'_2$. Thus, by definition of cross product we derive that $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) = \mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) = \langle\!|\, p'_1 \cap p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$. Moreover, from $p'_1 \preccurlyeq p_1$ and $p'_2 \preccurlyeq p_2$, we have that $p'_1 \cap p'_2 \preccurlyeq p_1 \cap p_2$. Therefore $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) = \langle\!|\, p'_1 \cap p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for $p'_1 \cap p'_2 \preccurlyeq p_1 \cap p_2 \equiv p$, as required.

- Let us prove ii). As $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\alpha} \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) = \langle\!|\, p'_1 \cap p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ and by Lemma 4, we deduce that $[\![p'_1 \cap p'_2]\!] \neq \emptyset$ and so there exists $t'' \in [\![p'_1 \cap p'_2]\!]$. Again, by Lemma 4, we have that $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\alpha} \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{t''} \mathsf{E}'$, for some $\mathsf{E}'$, with $\alpha \cdot t''$ prefix of some trace in $[\![p_1 \cap p_2]\!]$, as required.

*Case* $e = e_1 \cap e_2$, for some $e_1, e_2 \in \mathbb{P}\mathrm{rop}\mathbb{G}$. This case can be proved with a reasoning similar to that of the case $p_1 \cap p_2$.

*Inductive case:* $n > 1$, for $n \in \mathbb{N}$. Suppose $\mathsf{go} \bowtie P \xrightarrow{t} \mathsf{go} \bowtie J$ such that $t$ is a prefix of some trace in $[\![e]\!]$. Since $n > 1$, $\mathsf{go} \bowtie P \xrightarrow{t'} \mathsf{go} \bowtie J' \xrightarrow{\alpha} \mathsf{go} \bowtie J$ for some trace $t'$ such that $t = t' \cdot \alpha$. Since $t$ is a prefix of some trace in $[\![e]\!]$ it follows that $t'$ is a prefix of some trace in $[\![e]\!]$ as well. Hence, by inductive hypothesis, we have that:

1. $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{t'} \mathsf{E}' \bowtie J'$ in which either $\mathsf{E}' = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}' = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p' \in \mathbb{P}\mathrm{rop}\mathbb{L}$ such that $p' \preccurlyeq e$ and some automaton variable $\mathsf{X}$.
2. There is a trace $t'' \in [\![p']\!]$ such that $t' \cdot t''$ is a prefix of some trace in $[\![e]\!]$.

It remains to prove that if $\mathsf{go} \bowtie J' \xrightarrow{\alpha} \mathsf{go} \bowtie J$ such that $\alpha$ is a prefix of some trace in $[\![p']\!]$ then $\mathsf{E}' \bowtie J' \xrightarrow{\alpha} \mathsf{E} \bowtie J$. For that we resort to the proof of the base case, for $n = 1$. □

In order to prove Theorem 2 we need a couple of technical lemmata.

**Lemma 5** (Soundness of the synthesis)**.** *Let $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ be a global property and $\mathcal{P}$ be a set of observable actions such that* $\mathsf{events}(e) \subseteq \mathcal{P}$. *Let $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \mathsf{E}$ be an arbitrary execution trace of the synthesised automaton $\langle\!|\, e \,|\!\rangle^{\mathcal{P}}$. Then,*

1. *for* $t = enfAct(\lambda_1) \cdot \ldots \cdot enfAct(\lambda_n)$ *the trace* $\hat{t}$ *is a prefix of some trace in* $[\![e]\!]$;
2. *either* $\mathsf{E} = \langle\!\langle\ p'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ *or* $\mathsf{E} = \mathsf{Z}$, *with* $\mathsf{Z} = \langle\!\langle\ p'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, *for some* $p' \in \mathbb{PROP}$ *such that* $p' \preccurlyeq e$, *and some automaton variable* $\mathsf{X}$.

*Proof.* We proceed by induction on the length of the execution trace $\langle\!\langle\ e\ \rangle\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \mathsf{E}$. *Base case:* $n = 1$. In this case, $\langle\!\langle\ e\ \rangle\!\rangle^{\mathcal{P}} \xrightarrow{\lambda} \mathsf{E}$. We proceed by induction on the structure of $e$.

*Case* $e \equiv p^*$, for some $p \in \mathbb{PROP}$. We prove by induction on the structure of $p$ the following two results: i) for $\alpha = enfAct(\lambda)$, $\hat{\alpha}$ is a prefix of some trace in $[\![p]\!]$, and ii) either $\mathsf{E} = \langle\!\langle\ p'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}'}$ or $\mathsf{E} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!\langle\ p'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}'}$, for some $p' \in \mathbb{PROP}$ such that $p' \preccurlyeq p$ and some automaton variables $\mathsf{X}'$. As $\langle\!\langle\ p^*\ \rangle\!\rangle^{\mathcal{P}} \triangleq \mathsf{X}$, for $\mathsf{X} = \langle\!\langle\ p\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, results i) and ii) imply the required results (1) and (2), for $e = p^*$. We show the cases $p \equiv p_1; p_2$ and $p \equiv p_1 \cap p_2$, the others cases are similar or simpler.

Let $p \equiv p_1; p_2$ and $\langle\!\langle\ p_1; p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda} \mathsf{E}$. We prove the two results i) and ii) for $p_1 \neq \epsilon$, the case $p_1 = \epsilon$ is simpler. By definition, $\langle\!\langle\ p_1; p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ returns $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$, for $\mathsf{Z}' = \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, and $\mathsf{Z}' \neq \mathsf{X}$. As a consequence, from $p_1 \neq \epsilon$ and $\langle\!\langle\ p_1; p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda} \mathsf{E}$ it follows that $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda} \mathsf{E}_1$, for some $\mathsf{E}_1$.

- Let us prove i). Since $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda} \mathsf{E}_1$, by inductive hypothesis we have that $\hat{\alpha}$ is a prefix of some trace in $[\![p_1]\!]$. Thus, $\hat{\alpha}$ is a prefix of some trace in $[\![p_1; p_2]\!]$, as required.
- Let us prove ii). Again, since $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda} \mathsf{E}_1$, by inductive hypothesis either $\mathsf{E}_1 = \langle\!\langle\ p_1'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ or $\mathsf{E}_1 = \mathsf{Z}_1$, with $\mathsf{Z}_1 = \langle\!\langle\ p_1'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$, for some $p_1' \in \mathbb{PROP}$ such that $p_1' \preccurlyeq p_1$ and some automaton variables $\mathsf{Z}'$. Let us analyse $\mathsf{E}_1 = \langle\!\langle\ p_1'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ (the case $\mathsf{E}_1 = \mathsf{Z}_1$, with $\mathsf{Z}_1 = \langle\!\langle\ p_1'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$, is similar). As $\mathsf{E}_1 = \langle\!\langle\ p_1'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$ with $\mathsf{Z}' = \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ and $p_1' \preccurlyeq p_1$, by definition of the synthesis algorithm it follows that $\mathsf{E}_1 = \langle\!\langle\ p_1'; p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for $p_1'; p_2 \preccurlyeq p_1; p_2 \equiv p$, as required.

Let $p \equiv p_1 \cap p_2$ and $\langle\!\langle\ p_1 \cap p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda} \mathsf{E}$. By definition, the synthesis algorithm applied to $\langle\!\langle\ p_1 \cap p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ returns $\mathsf{Prod}_{\mathsf{X}}(\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}})$. Let us prove the results i) and ii).

- Result i) follows directly from Lemma 4.
- Let us prove ii). Let us consider the transitions of $\mathsf{Prod}_{\mathsf{X}}(\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}})$. By inspection of the cross product in Table 11.1, the most interesting case is when $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \sum_{i \in I} \lambda_i.\mathsf{E}_i$ and $\langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \sum_{j \in J} \nu_j.\mathsf{E}_j$. In this case, the cross product $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\sum_{i \in I} \lambda_i.\mathsf{E}_i, \sum_{j \in J} \nu_j.\mathsf{E}_j)$ is equal to:

$$\begin{cases} \sum\limits_{\alpha \in \mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}} {}^{-}\alpha.\mathsf{X} & \text{if } H = \emptyset \\ \sum\limits_{(i,j) \in H} (\lambda_i.\mathsf{X}_{i,j} + \lambda_i \prec \mathsf{end}.\mathsf{X}_{i,j}) + \sum\limits_{\alpha \in (\mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}) \setminus \bigcup_{(i,j) \in H} \lambda_i} {}^{-}\alpha.\mathsf{X} & \text{if } H \neq \emptyset \\ \text{for } \mathsf{X}_{i,j} = \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) \end{cases}$$

with $H = \{(i,j) \in I \times J : \lambda_i = \nu_j \in \mathcal{P} \text{ and } \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\mathsf{E}_i, \mathsf{E}_j) \neq \sum_{\alpha \in \mathcal{P} \setminus \{\mathsf{tick},\mathsf{end}\}} {}^{-}\alpha.\mathsf{X}\}$. The first case satisfies result ii) as by definition $\mathsf{X} = \langle\!\langle\ p_1 \cap p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ and $p_1 \cap p_2 \preccurlyeq p_1 \cap p_2 \equiv p$.

In the second case, the edit automaton has following three (families of) transitions:

(a) $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\lambda_i} \mathsf{X}_{i,j}$, for $(i,j) \in H$;

(b) $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\lambda_i \prec \mathsf{end}} \mathsf{X}_{i,j}$, for $(i,j) \in H$;

(c) $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{{}^{-}\alpha} \mathsf{X}$, for $\alpha \in (\mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}) \setminus \bigcup_{(i,j)\in H} \lambda_i$.

We prove the result for the case (a); cases (b) and (c) can be proved in a similar manner. By definition of cross product, it holds that $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda_i} \mathsf{E}_i$ and $\langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda_i} \mathsf{E}_j$. From $\langle\!\langle\ p_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda_i} \mathsf{E}_i$, by inductive hypothesis we have that either $\mathsf{E}_i = \langle\!\langle\ p'_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}_i = \mathsf{Z}_1$, with $\mathsf{Z}_1 = \langle\!\langle\ p'_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p'_1 \in \mathbb{Prop}\mathbb{L}$ such that $p'_1 \preccurlyeq p_1$. Similarly, from $\langle\!\langle\ p_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}} \xrightarrow{\lambda_i} \mathsf{E}_j$, by inductive hypothesis we have that either $\mathsf{E}_j = \langle\!\langle\ p'_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}_j = \mathsf{Z}_2$, with $\mathsf{Z}_2 = \langle\!\langle\ p'_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p'_2 \in \mathbb{Prop}\mathbb{L}$ such that $p'_2 \preccurlyeq p_2$. Therefore, by definition of cross product, we derive that $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) = \mathsf{Prod}_{\mathsf{X}}(\langle\!\langle\ p'_1\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\langle\ p'_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}) = \langle\!\langle\ p'_1 \cap p'_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$. Moreover, since $p'_1 \preccurlyeq p_1$ and $p'_2 \preccurlyeq p_2$, we have that $p'_1 \cap p'_2 \preccurlyeq p_1 \cap p_2$. Therefore, $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) = \langle\!\langle\ p'_1 \cap p'_2\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for $p'_1 \cap p'_2 \preccurlyeq p_1 \cap p_2 \equiv p$, as required.

*Case* $e = e_1 \cap e_2$ for some $e_1, e_2 \in \mathbb{Prop}\mathbb{G}$. This case can be proved with a reasoning similar to that seen in the proof of case $p_1 \cap p_2$.

*Inductive case:* $n > 1$, for $n \in \mathbb{N}$. Suppose $\langle\!\langle\ e\ \rangle\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} \mathsf{E}' \xrightarrow{\lambda_n} \mathsf{E}$, for $n > 1$. $\langle\!\langle\ e\ \rangle\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} \mathsf{E}' \xrightarrow{\lambda_n} \mathsf{E}$. Thus, by induction, we have that:

1. for $t' = \mathit{enfAct}(\lambda_1) \cdot \ldots \cdot \mathit{enfAct}(\lambda_{n-1})$ the trace $\widehat{t'}$ is a prefix of some trace in $[\![e]\!]$, and
2. either $\mathsf{E}' = \langle\!\langle\ p'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}' = \mathsf{Z}$, with $\mathsf{Z} = \langle\!\langle\ p'\ \rangle\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p' \in \mathbb{Prop}\mathbb{L}$ such that $p' \preccurlyeq e$ and some automaton variables $\mathsf{Z}$ and $\mathsf{X}$.

To conclude the proof it is sufficient to prove that given $\mathsf{E}' \xrightarrow{\lambda_n} \mathsf{E}$ and $\alpha_n = \mathit{enfAct}(\lambda_n)$, it holds that $\hat{\alpha_n}$ is a prefix of some trace in $[\![p']\!]$. For that we resort to the proof of the base case. □

In the next lemma, we prove that, given the execution traces of a monitored controller, we can always extract from them the traces performed by its edit automaton and its monitored controller in isolation. For proving that we need a technical definition. Let $\lambda \in \{\alpha, \alpha \prec \beta, {}^{-}\alpha\}$ be an action for an edit automaton, we write $\mathit{ctrlAct}(\lambda)$ to denote the controller action associated to $\lambda$. Formally, $\mathit{ctrlAct}(\alpha) = \mathit{ctrlAct}({}^{-}\alpha) = \alpha$ and $\mathit{ctrlAct}(\alpha \prec \beta) = \beta$.

**Lemma 6** (Trace decomposition). *Let $e \in \mathbb{Prop}\mathbb{G}$ be a global property, $P \in \mathbb{Ctrl}$ be a controller, and $\mathcal{P}$ be the set of all possible actions of $P$ such that* $\mathsf{events}(e) \subseteq \mathcal{P}$. *Then, for any execution trace $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{\alpha_1} \mathsf{E}_1 \bowtie J_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \mathsf{E}_n \bowtie J_n$ it hold the following results:*

1. $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \mathsf{E}_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} \mathsf{E}_n$, *with* $\alpha_i = enfAct(\lambda_i)$;
2. *for* $J_0 = P$ *and* $1 \leq i \leq n$, *either* $J_{i-1} \xrightarrow{\beta_i} J_i$, *with* $\beta_i = ctrlAct(\lambda_i) \in \mathcal{P}$, *or* $J_i = J_{i-1}$.

*Proof.* The proof is by induction on the length $n$ of the execution trace $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{\alpha_1} \mathsf{E}_1 \bowtie J_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \mathsf{E}_n \bowtie J_n$. The base case, for $n = 1$, is trivial since if $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{\alpha} \mathsf{E} \bowtie J$ then $\alpha = \mathsf{tick}$ and the synthesis algorithm in Table 7.5 never return an edit automaton suppressing a tick-action. Hence, we focus on the inductive step. Let $n > 1$ and $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{\alpha_1} \mathsf{E}_1 \bowtie J_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} \mathsf{E}_n \bowtie J_n$. By inductive hypothesis we have:

- $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \mathsf{E}_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_{n-1}} \mathsf{E}_{n-1}$, with $\alpha_i = enfAct(\lambda_i)$;
- for $J_0 = P$ and $1 \leq i \leq n-1$, either $J_{i-1} \xrightarrow{\beta_i} J_i$, with $\beta_i = ctrlAct(\lambda_i) \in \mathcal{P}$, or $J_i = J_{i-1}$.

Thus, we have to prove the following results: 1) $\mathsf{E}_{n-1} \xrightarrow{\lambda_n} \mathsf{E}_n$ with $\alpha_n = enfAct(\lambda_n)$; 2) either $J_{n-1} \xrightarrow{\beta_n} J_n$ or $J_n = J_{n-1}$, with $\beta_n = ctrlAct(\lambda_n)$. Let us consider the step $\mathsf{E}_{n-1} \bowtie J_{n-1} \xrightarrow{\alpha_n} \mathsf{E}_n \bowtie J_n$. By an application of Lemma 5 we have that either $\mathsf{E}_{n-1} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E}_{n-1} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p' \in \mathbb{Prop}\mathbb{L}$ such that $p' \preccurlyeq e$ and some automaton variable $\mathsf{X}$. We consider the case $\mathsf{E}_{n-1} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ (the case $\mathsf{E}_{n-1} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, is similar). We proceed by case analysis on the structure of $p'$. We show the case $p' \equiv p'_1 \cap p'_2$, the others cases are similar or simpler.

Case $p' \equiv p'_1 \cap p'_2$. We have that $\mathsf{E}_{n-1} = \langle\!|\, p'_1 \cap p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$. We prove the results (1) and (2). Let us consider the transitions of $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$. By definition of cross product in Table 11.1, the most interesting case is when $\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \sum_{i \in I} \lambda_i.\mathsf{E}_i$ and $\langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \sum_{j \in J} \nu_j.\mathsf{E}_j$. In this case, $\mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\sum_{i \in I} \lambda_i.\mathsf{E}_i, \sum_{j \in J} \nu_j.\mathsf{E}_j)$ is equal to:

$$\begin{cases} \sum\limits_{\alpha \in \mathcal{P} \setminus \{\mathsf{tick},\mathsf{end}\}} {}^{-}\alpha.\mathsf{X} & \text{if } H = \emptyset \\ \sum\limits_{(i,j) \in H} (\lambda_i.\mathsf{X}_{i,j} + \lambda_i \prec \mathsf{end}.\mathsf{X}_{i,j}) + \sum\limits_{\alpha \in (\mathcal{P} \setminus \{\mathsf{tick},\mathsf{end}\}) \setminus \bigcup_{(i,j) \in H} \lambda_i} {}^{-}\alpha.\mathsf{X} & \text{if } H \neq \emptyset \\ \text{for } \mathsf{X}_{i,j} = \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}_{i,j}}(\mathsf{E}_i, \mathsf{E}_j) \end{cases}$$

with $H = \{(i,j) \in I \times J : \lambda_i = \nu_j \in \mathcal{P} \text{ and } \mathsf{Prod}^{\mathcal{P}}_{\mathsf{X}}(\mathsf{E}_i, \mathsf{E}_j) \neq \sum_{\alpha \in \mathcal{P} \setminus \{\mathsf{tick},\mathsf{end}\}} {}^{-}\alpha.\mathsf{X}\}$. In order to prove the results (1) and (2), we proceed by case analysis on the transitions of $J_{n-1}$. We have three cases.

i) Let $J_{n-1} \xrightarrow{\beta_n} J_n$, for $\beta_n \in \mathsf{events}(e) \subseteq \mathcal{P}$ and $\beta_n = \lambda_i$ for $(i,j) \in H$. Result (1) follows because $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\beta_n} \mathsf{X}_{i,j}$ with $enfAct(\beta_n) = \beta_n$. Moreover, by an application of rule (Allow) we derive the transition $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p'_1 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p'_2 \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \bowtie J_{n-1} \xrightarrow{\beta_n} \mathsf{X}_{i,j} \bowtie J_n$, with $ctrlAct(\beta_n) = \beta_n$. This implies the result (2).

ii) Let $J_{n-1} \xrightarrow{\mathsf{end}} J_n$. As $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p_1' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{\beta_n \prec \mathsf{end}} \mathsf{X}_{i,j}$, for $\beta_n \in \mathsf{events}(e) \subseteq \mathcal{P}$ and $\beta_n = \lambda_i$ for $(i,j) \in H$, by an application of rule (Insert) the monitored controller has the transition $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p_1' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \bowtie J_{n-1} \xrightarrow{\beta_n} \mathsf{X}_{i,j} \bowtie J_n$, with $\mathit{enfAct}(\beta_n \prec \mathsf{end}) = \beta_n$. This implies the result (1). Moreover, $J_n = J_{n-1}$ and $\mathit{ctrlAct}(\beta_n \prec \mathsf{end}) = \mathsf{end}$. This implies result (2).

iii) Let $J_{n-1} \xrightarrow{\beta_n} J_n$, for $\beta_n \in (\mathcal{P} \setminus \{\mathsf{tick}, \mathsf{end}\}) \setminus \bigcup_{(i,j)\in H} \lambda_i$. Since we have the transition $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p_1' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \xrightarrow{{}^{-}\beta_n} \mathsf{Z}$ with $\mathit{enfAct}({}^{-}\beta_n) = \tau$ we derive the result (1). Moreover, by an application of the rule (Suppress), we have the transition $\mathsf{Prod}_{\mathsf{X}}(\langle\!|\, p_1' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!|\, p_2' \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \bowtie J_{n-1} \xrightarrow{\tau} \mathsf{X} \bowtie J_n$ with $\mathit{ctrlAct}({}^{-}\beta_n) = \beta_n$. This implies (2). □

*Proof of Theorem 2 (Soundness).* Let $t = \alpha_1 \cdot \ldots \cdot \alpha_n$ be a trace such that $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{t} \mathsf{E} \bowtie J$, for some $\mathsf{E} \in \mathbb{E}\mathrm{dit}$ and some controller $J$. By an application of Lemma 6 there exist $\mathsf{E}_i \in \mathbb{E}\mathrm{dit}$ and $\lambda_i$, for $1 \leq i \leq n$, such that: $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \mathsf{E}_1 \xrightarrow{\lambda_2} \ldots \xrightarrow{\lambda_n} \mathsf{E}_n = \mathsf{E}$, with $\alpha_i = \mathit{enfAct}(\lambda_i)$. Thus, $t = \mathit{enfAct}(\lambda_1) \cdot \ldots \cdot \mathit{enfAct}(\lambda_n)$. By Lemma 5, $\hat{t}$ is a prefix of some trace in $[\![e]\!]$, as required. □

**Lemma 7** (Deadlock-freedom of the synthesis)**.** *Let $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ be a global property and $\mathcal{P}$ be a set of observable actions s.t.* $\mathsf{events}(e) \subseteq \mathcal{P}$. *Then the edit automaton $\langle\!|\, e \,|\!\rangle^{\mathcal{P}}$ does not deadlock.*

*Proof.* Given an arbitrary execution $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} \mathsf{E}$, the proof is by induction on the length $n$ of the execution trace. By an application of Lemma 5 we have that either $\mathsf{E} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for $p \in \mathbb{P}\mathrm{rop}\mathbb{L}$ and some automaton variable $\mathsf{X}$. Hence, the result follows by inspection of the synthesis function of Table 7.5 and by induction on the structure of $p$. □

*Proof of Theorem 3 (Deadlock-freedom).* Let $t$ be a trace such that $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{t} \mathsf{E} \bowtie J$, for some edit automaton $\mathsf{E}$ and controller $J$. By contradiction we assume that $\mathsf{E} \bowtie J$ is in deadlock. Notice that, by definition, our controllers $J$ never deadlock. By Lemma 7 the automaton $\langle\!|\, e \,|\!\rangle^{\mathcal{P}}$ never deadlock as well. Consequently, we have that for any transition $J \xrightarrow{\alpha} J'$ there is no action $\lambda$ for $\mathsf{E}$, such that the monitored controller $\mathsf{E} \bowtie J$ may progress according to one of the rules: (Allow), (Suppress) and (Insert). Now, let us consider the class of edit automata with the following form:

$$\sum_{i\in I} \alpha_i.\mathsf{X}_i + \sum_{i\in I, \alpha_i \neq \mathsf{end}} \alpha_i \prec \mathsf{end}.\mathsf{X}_i + \sum_{\alpha \in \mathcal{P}\setminus(\cup_{i\in I}\alpha_i \cup \{\mathsf{tick},\mathsf{end}\})} {}^{-}\alpha.\mathsf{X}$$

This class of edit automata, denoted with $\mathcal{E}$, may only deadlock the enforcement when the controller may only perform tick-actions. We now show that either $\mathsf{E} \in \mathcal{E}$ or $\mathsf{E} = \mathsf{Y}$, with $\mathsf{Y} = \mathsf{E}'$ and $\mathsf{E}' \in \mathcal{E}$. By an application of Lemma 5, we have that either $\mathsf{E} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ or $\mathsf{E} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, for some $p \in \mathbb{P}\mathrm{rop}\mathbb{L}$ and some automaton variable $\mathsf{X}$. We consider the case $\mathsf{E} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ (the case $\mathsf{E} = \mathsf{Z}$, with $\mathsf{Z} = \langle\!|\, p \,|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, is similar). We proceed

by induction on the structure of $p$ and we give the cases $p_1; p_2$ and $p_1 \cap p_2$; the other cases are simpler.

Let $p \equiv p_1; p_2$. By definition, $\langle\!\|\ p_1; p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$ returns $\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{Z}'}$, for $\mathsf{Z}' = \langle\!\|\ p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$, and $\mathsf{Z}' \neq \mathsf{X}$. By inductive hypothesis we have either $\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{Z}'} \in \mathcal{E}$ or $\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \mathsf{Y}_1$, with $\mathsf{Y}_1 = \mathsf{E}'_1$ and $\mathsf{E}'_1 \in \mathcal{E}$. In the first case, $\mathsf{E} = \langle\!\|\ p\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \langle\!\|\ p_1; p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} \in \mathcal{E}$, as required; the second case is similar.

Let $p \equiv p_1 \cap p_2$. By definition, $\langle\!\|\ p_1 \cap p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \mathsf{Prod}_{\mathsf{X}}(\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\|\ p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}})$. By inductive hypothesis either $\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} \in \mathcal{E}$ or $\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \mathsf{Y}_1$, with $\mathsf{Y}_1 = \mathsf{E}'_1$ and $\mathsf{E}'_1 \in \mathcal{E}$; similarly. Still by inductive hypothesis either $\langle\!\|\ p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} \in \mathcal{E}$ or $\langle\!\|\ p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \mathsf{Y}_2$, with $\mathsf{Y}_2 = \mathsf{E}'_2$ and $\mathsf{E}'_2 \in \mathcal{E}$. By definition of cross product (see Table 11.1), it follows that $\mathsf{E} = \langle\!\|\ p\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \langle\!\|\ p_1 \cap p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \mathsf{Prod}_{\mathsf{X}}(\langle\!\|\ p_1\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}, \langle\!\|\ p_2\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}) \in \mathcal{E}$.

Now, since $\mathsf{E} \in \mathcal{E}$ but $\mathsf{E} \bowtie J$ is in deadlock, it follows that $J$ may only perform tick-actions and $\mathsf{E} \xrightarrow{\text{tick}}\!\!\!\!\!\!/\ \ $. From the first fact, we derive $J = \mathsf{tick}^h.S$, for $0 < h \leq k$. Since tick-actions cannot be suppressed by edit automata in $\mathcal{E}$, we have that $t = t' \cdot \mathsf{tick}^{k-h}$, for some possibly empty trace $t'$ terminating with an end. By Theorem 2, $t = t' \cdot \mathsf{tick}^{k-h} \in [\![e]\!]$. And since $e$ is $k$-sleeping we derive $p = \mathsf{tick}^h.p'$, for some $p'$. Since $\langle\!|\, e \,|\!\rangle^{\mathcal{P}}$ is sound (Lemma 5) we derive that $\mathsf{E} = \langle\!\|\ p\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}} = \langle\!\|\ \mathsf{tick}^h.p'\ \|\!\rangle^{\mathcal{P}}_{\mathsf{X}}$. Finally, $h > 0$ implies $\mathsf{E} \xrightarrow{\text{tick}} \mathsf{E}'$, for some $\mathsf{E}'$, in contradiction with what stated four lines above. □

*Proof of Theorem 4 (Divergence-freedom).* Let $e \in \mathbb{P}\mathrm{rop}\mathbb{G}$ be a global property in its general form, given by the intersection of $n \geq 1$ global properties $p^*_1 \cap \cdots \cap p^*_n$, for $p_i \in \mathbb{P}\mathrm{rop}\mathbb{L}$, with $1 \leq i \leq n$. As $e$ is well-formed, according to Definition 4 also all local properties $p_i$ are well-formed. This means that they all terminate with an end event. Thus, in all global properties $p^*_i$, for $1 \leq i \leq n$, the number of events within two subsequent end events is always finite. The same holds for the property $e$. Now, let $t$ be an arbitrary trace such that $\langle\!|\, e \,|\!\rangle^{\mathcal{P}} \bowtie P \xrightarrow{t} \mathsf{E} \bowtie J$, for some edit automaton $\mathsf{E}$ and controller $J$. And let $k = \max_{1 \leq i \leq n} k_i$, where $k_i$ is the length of the longest trace of $[\![p_i]\!]$, for $1 \leq i \leq n$. Thus, if $\mathsf{E} \bowtie J \xrightarrow{t'} \mathsf{E}' \bowtie J'$, with $\mid t' \mid \geq k$, and since by Theorem 2 we have that $t \cdot t'$ is a prefix of some trace $[\![e]\!]$, then $\mathsf{end} \in t'$. □

# Bibliography


[1] M. Abadi, B. Blanchet, and C. Fournet. "The Applied Pi Calculus: Mobile Values, New Names, and Secure Communication". In: *Journal of the ACM* 65.1 (2018), 1:1–1:41.

[2] M. Abadi and A. D. Gordon. "A Calculus for Cryptographic Protocols: The Spi Calculus". In: *ACM CCS*. ACM, 1997, pp. 36–47.

[3] A. Abbasi and M. Hashemi. "Ghost in the PLC Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack". In: *Black Hat*. 2016, pp. 1–35.

[4] L. Aceto et al. "Adventures in monitorability: from branching to linear time and back again". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 52:1–52:29.

[5] L. Aceto et al. "On Runtime Enforcement via Suppressions". In: *CONCUR*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 34:1–34:17.

[6] Luca Aceto et al. "Determinizing monitors for HML with recursion". In: *Journal of Logical and Algebraic Methods in Programming* 111 (2020), p. 100515.

[7] Luca Aceto et al. "On Bidirectional Runtime Enforcement". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Cham: Springer International Publishing, 2021, pp. 3–21.

[8] T. Alladi, V. Chamola, and S. Zeadally. "Industrial Control Systems: Cyberattack trends and countermeasures". In: *Computer Communications* 155 (2020), pp. 1 –8.

[9] R. Alur and D. L. Dill. "A theory of timed automata". In: *Theoretical Computer Science* 126.2 (1994), pp. 183 –235.

[10] R. Alur and P. Černý. "Streaming Transducers for Algorithmic Verification of Single-Pass List-Processing Programs". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. ACM, 2011, 599–610.

[11] R. Alves T. et al. "OpenPLC: An open source alternative to automation". In: *GHTC 2014*. 2014, pp. 585–589.

[12] R. Anderson and S. Fuloria. "Security economics and critical national infrastructure". In: *Economics of Information Security and Privacy*. Springer, 2010, pp. 55–66.

[13] O Andreeva et al. *Industrial Control Systems Vulnerabilities Statistics-Kaspersky Lab*. 2019. URL: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2016/07/07190426/KL_REPORT_ICS_Statistic_vulnerabilities.pdf.

[14] M. Antonakakis et al. "Understanding the mirai botnet". In: *26th {USENIX} security symposium ({USENIX} Security 17)*. 2017, pp. 1093–1110.

[15] D. Antonioli and N. O. Tippenhauer. "MiniCPS: A Toolkit for Security Research on CPS Networks". In: *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy*. CPS-SPC '15. New York, NY, USA: Association for Computing Machinery, 2015, 91–100.

[16] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[17] E. Bartocci et al. "Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications". In: *Lectures on Runtime Verification - Introductory and Advanced Topics*. Vol. 10457. LNCS. Springer, 2018, pp. 135–175.

[18] Danièle Beauquier, Joëlle Cohen, and Ruggero Lanotte. "Security policies enforcement using finite and pushdown edit automata". In: *International Journal of Information Security* 12.4 (2013), pp. 319–336.

[19] G. Behrmann et al. "UPPAAL 4.0". In: *QEST 2006*. IEEE Computer Society, 2006, pp. 125–126.

[20] G. Bernieri et al. "Monitoring system reaction in cyber-physical testbed under cyber-attacks". In: *Computers  Electrical Engineering* 59 (2017), pp. 86 –98.

[21] J. Berstel. *Transductions and context-free languages*. Springer-Verlag, 2013.

[22] M. Bielova. "A theory of constructive and predictable runtime enforcement mechanisms". PhD thesis. University of Trento, 2011.

[23] N. Bielova and F. Massacci. "Predictability of Enforcement". In: *Engineering Secure Software and Systems*. 2011, pp. 73–86.

[24] B. Blanchet. "Automatic Verification of Correspondences for Security Protocols". In: *Journal of Computer Security* 17.4 (2009), pp. 363 –434.

[25] K. Boeckl et al. *Considerations for managing Internet of Things (IoT) cybersecurity and privacy risks*. US Department of Commerce, National Institute of Standards and Technology, 2019.

[26] S. Bogomolov et al. "Assume-Guarantee Abstraction Refinement Meets Hybrid Systems". In: *HVC 2014*. Ed. by Eran Yahav. Vol. 8855. LNCS. Springer, 2014, pp. 116–131. DOI: 10.1007/978-3-319-13338-6.

[27] W. M. Brandin B. A. Wonham. "Supervisory control of timed discrete-event systems". In: *IEEE Transactions on Automatic Control* 39.2 (1994), pp. 329–342.

[28] Carlos E. Budde et al. "A Statistical Model Checker for Nondeterminism and Rare Events". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer International Publishing, 2018, pp. 340–358.

[29] E. Byres and P. Eng. "Using ANSI/ISA-99 standards to improve control system security". In: *White paper, Tofino Security* (2012).

[30] L. Cardelli and A. Gordon. "Mobile Ambients". In: *TCS* 240.1 (2000), pp. 177–213.

[31] A.A. Cárdenas et al. "Attacks against process control systems: risk assessment, detection, and response". In: *ASIACCS*. 2011, pp. 355–366.

[32] Ian Cassar. "Developing Theoretical Foundations for Runtime Enforcement". PhD thesis. University of Malta and Reykjavik University, 2020.

[33] Ian Cassar and Adrian Francalanza. "Runtime Adaptation for Actor Systems". In: *Runtime Verification*. Springer International Publishing, 2015, pp. 38–54.

[34] Ian Cassar et al. "A Suite of Monitoring Tools for Erlang". In: *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*. Vol. 3. EasyChair, 2017, pp. 41–47.

[35] C. Cheh et al. "Determining Tolerable Attack Surfaces that Preserves Safety of Cyber-Physical Systems". In: *PRDC*. IEEE Computer Society, 2018, pp. 125–134.

[36] X. Chen, E. Ábrahám, and S. Sankaranarayanan. "Flow*: An Analyzer for Non-linear Hybrid Systems". In: *CAV 2013*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. LNCS. Springer, 2013, pp. 258–263. DOI: 10.1007/978-3-642-39799-8.

[37] Anton Cherepanov. "WIN32/INDUSTROYER: A new threat for industrial control systems". In: *White paper, ESET (June 2017)* (2017).

[38] S. Cheung et al. "Using model-based intrusion detection for SCADA networks". In: Citeseer.

[39] Y. S. Chow and H. Robbins. "On the Asymptotic Theory of Fixed-Width Sequential Confidence Intervals for the Mean". In: *The Annals of Mathematical Statistics* 36.2 (1965), pp. 457–462.

[40] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 8.2 (1986), pp. 244–263.

[41] Edmund M Clarke et al. *Handbook of model checking*. Vol. 10. Springer, 2018.

[42] L. F. Cómbita et al. "Response and reconfiguration of cyber-physical control systems: A survey". In: *CCAC*. IEEE. 2015, pp. 1–6.

[43] *Cyber Incident Blamed for Nuclear Power*.

[44] D. David et al. "Time for Statistical Model Checking of Real-Time Systems". In: *CAV 2011*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 349–355. DOI: 10.1007/978-3-642-22110-1.

[45] R. C. Dorf and R. H. Bishop. *Modern control systems*. Pearson, 2011.

[46] Z. Drias, Serhrouchni A., and O. Vogel. "Analysis of cyber security for industrial control systems". In: *2015 International Conference on Cyber Security of Smart Cities, Industrial Control System and Communications (SSIC)*. 2015, pp. 1–8.

[47] P. S. Duggirala et al. "C2E2: A Verification Tool for Stateflow Models". In: *TACAS 2015*. Ed. by Christel Baier and Cesare Tinelli. Vol. 9035. LNCS. Springer, 2015, pp. 68–82. DOI: 10.1007/978-3-662-46681-0.

[48] M. Fabian and A. Hellgren. "PLC-based implementation of supervisory control for discrete event systems". In: *37th IEEE CDC*. Vol. 3. 1998, pp. 3305–3310.

[49] Y. Falcone, J-C. Fernandez, and L. Mounier. "What can you verify and enforce at runtime?" In: *International Journal on Software Tools for Technology Transfer* 14.3 (2012), pp. 349–382.

[50] Y. Falcone, K. Havelund, and G. Reger. "A Tutorial on Runtime Verification". In: *EDSS*. Vol. 34. NATO Science for Peace and Security Series IOS Press, 2013, pp. 141–175.

[51] Y. Falcone et al. "Runtime enforcement monitors: composition, synthesis, and enforcement abilities". In: *Formal Methods in System Design* 38.3 (2011), pp. 223–262.

[52] A. Francalanza. "A theory of monitors". In: *Information and Computation* (2021), p. 104704.

[53] A. Francalanza et al. "A Foundation for Runtime Monitoring". In: *RV*. Vol. 10548. LNCSSpringer, 2017, pp. 8–29.

[54] Adrian Francalanza. "Consistently-Detecting Monitors". In: *28th International Conference on Concurrency Theory (CONCUR 2017)*. Vol. 85. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 8:1–8:19.

[55] Adrian Francalanza and Jasmine Xuereb. "On Implementing Symbolic Controllability". In: *Coordination Models and Languages*. Springer International Publishing, 2020, pp. 350–369.

[56] G. Frehse. "PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech". In: *International Journal on Software Tools for Technology Transfer* 10.3 (2008), pp. 263–279.

[57] G. Frehse. *PHAVer Language Overview v0.35*. 2006. URL: http://www-verimag.imag.fr/~frehse/phaver_web/phaver_lang.pdf.

[58] G. Frehse et al. "A Toolchain for Verifying Safety Properties of Hybrid Automata via Pattern Templates". In: *ACC*. 2018, pp. 2384–2391.

[59] G. Frehse et al. "SpaceEx: Scalable Verification of Hybrid Systems". In: *CAV 2011*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 379–395. DOI: 10.1007/978-3-642-22110-1.

[60] Warren Gay. *Mastering the raspberry PI*. Apress, 2014.

[61] J. Giraldo et al. "A Survey of Physics-Based Attack Detection in Cyber-Physical Systems". In: *ACM Comput. Surv.* 51.4 (2018), 76:1–76:36.

[62] J. Giraldo et al. "Hide and Seek: An Architecture for Improving Attack-Visibility in Industrial Control Systems". In: *Applied Cryptography and Network Security*. Springer International Publishing, 2019, pp. 175–195.

[63] J. Giraldo et al. "Security and Privacy in Cyber-Physical Systems: A Survey of Surveys". In: *IEEE Design Test* 34.4 (2017), pp. 7–17.

[64] J. Goh et al. "A Dataset to Support Research in the Design of Secure Water Treatment Systems". In: *CRITIS*. Vol. 10242. LNCS. Springer, 2017, pp. 88–99.

[65] D. Gollmann and M. Krotofil. "Cyber-Physical Systems Security". In: *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*. Vol. 9100. LNCS. Springer, 2016, pp. 195–204.

[66] Dieter Gollmann et al. "Cyber-Physical Systems Security: Experimental Analysis of a Vinyl Acetate Monomer Plant". In: *ACM CCPS*. 2015, pp. 1–12.

[67] N. Govil, A. Agrawal, and N. O. Tippenhauer. "On Ladder Logic Bombs in Industrial Control Systems". In: *SECPRE@ESORICS 2017*. Vol. 10683. LNCS. Springer, 2018, pp. 110–126.

[68] A. Greenberg. *Hackers remotely kill a jeep on the highway with me in it*. Wired, 2015.

[69] *Hack the planet with amerka GUI — Ultimate Internet of Things/Industrial Control Systems reconnaissance tool*. 2020. URL: https://www.offensiveosint.io/.

[70] D. Hadžiosmanović et al. "Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes". In: *Proceedings of the 30th Annual Computer Security Applications Conference*. ACSAC '14. Association for Computing Machinery, 2014, 126–135.

[71] D. Hadžiosmanović et al. "N-gram against the machine: On the feasibility of the n-gram network analysis for binary protocols". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2012, pp. 354–373.

[72] E. M. Hahn et al. "A compositional modelling and analysis framework for stochastic hybrid systems". In: *FMSD* 43.2 (2013), pp. 191–232.

[73] A. Hartmanns and H. Hermanns. "The Modest Toolset: An integrated environment for quantitative modelling and verification". In: *TACAS*. Vol. 8413. LNCS. Springer, 2014, pp. 593–598.

[74] W. K. Hastings. "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". In: *Biometrika* 57.1 (1970), pp. 97–109.

[75] M. Hennessy and T. Regan. "A Process Algebra for Timed Systems." In: *Information and Computation* 117.2 (1995), pp. 221–239.

[76] T. A. Henzinger. "The Theory of Hybrid Automata". In: *LICS*. IEEE Computer Society, 1996, pp. 278–292.

[77] T. A. Henzinger et al. "What's Decidable about Hybrid Automata?" In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 94 –124.

[78] L. Huang and E.-Y. Kang. "Formal Verification of Safety & Security Related Timing Constraints for a Cooperative Automotive System". In: *FASE*. Vol. 11424. LNCS. Springer, 2019, pp. 210–227.

[79] Y. Huang et al. "Understanding the physical and economic consequences of attacks on control systems". In: *IJCIP* 2.3 (2009), pp. 73–83.

[80] T. Huffmire et al. *Handbook of FPGA design security*. Springer Science & Business Media, 2010.

[81] K. H. Johansson. "The quadruple-tank process: A multivariable laboratory process with an adjustable zero". In: *IEEE Trans. on Control System Tech.* 8.3 (2000), pp. 456–465.

[82] K. H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. 2nd. Springer Publishing Company, Incorporated, 2010.

[83] B. Johnson et al. "Attackers deploy ICS attack framework "TRITON" and cause operational disruption to critical infrastructure". In: *Threat Research Blog* (2017).

[84] B. Könighofer et al. “Shield synthesis”. In: *Formal Methods in System Design* 51.2 (2017), pp. 332–361.

[85] M. Krotofil and A. A. Cárdenas. “Resilience of Process Control Systems to Cyber-Physical Attacks”. In: *NordSec*. Vol. 8208. LNCS. Springer, 2013, pp. 166–182.

[86] M. Krotofil et al. “Vulnerabilities of cyber-physical systems to stale data - Determining the optimal time to launch attacks”. In: *Int. J. Critical Infrastructure Protection* 7.4 (2014), pp. 213–232.

[87] R. Kumar and M. Stoelinga. “Quantitative Security and Safety Analysis with Attack-Fault Trees”. In: *HASE*. IEEE Computer Society, 2017, pp. 25–32.

[88] David Kushner. “The real story of STUXnet”. In: *IEEE Spectrum* 50.3 (2013), pp. 48 –53.

[89] M. Z. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *CAV 2011*. Vol. 6806. LNCS. Springer, 2011, pp. 585–591.

[90] G. Lafferriere, G. J. Pappas, and S. Sastry. “O-minimal hybrid systems”. In: *Mathematics of Control, Signals, and Systems* 13.1 (2000), pp. 1–21.

[91] R. Lanotte, M. Merro, and A. Munteanu. “A Modest Security Analysis of Cyber-Physical Systems: A Case Study”. In: *FORTE*. Vol. 10854. LNCS. Springer, 2018, pp. 58–78.

[92] R. Lanotte, M. Merro, and S. Tini. “A Probabilistic Calculus of Cyber-Physical Systems”. In: *Information and Computation* (2020).

[93] R. Lanotte, M. Merro, and S. Tini. “Towards a Formal Notion of Impact Metric for Cyber-Physical Attacks”. In: *IFM*. Vol. 11023. LNCS. Springer, 2018, pp. 296–315.

[94] R. Lanotte et al. “A Formal Approach to Cyber-Physical Attacks”. In: *CSF 2017*. IEEE Computer Society, 2017, pp. 436–450. DOI: 10.1109/CSF.2017.12.

[95] R. Lanotte et al. “A Formal Approach to Physics-based Attacks in Cyber-physical Systems”. In: *ACM Transactions on Privacy and Security* 23.1 (2020), 3:1–3:41.

[96] R. Lanotte et al. “A Formal Approach to Physics-based Attacks in Cyber-physical Systems”. In: *ACM Trans. Priv. Secur.* 23.1 (2020), 3:1–3:41.

[97] B. Lantz, B. Heller, and N. McKeown. “A Network in a Laptop: Rapid Prototyping for Software-Defined Networks”. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. ACM, 2010.

[98] Edward A Lee. “The past, present and future of cyber-physical systems: A focus on models”. In: *Sensors* 15.3 (2015), pp. 4837–4869.

[99] R. M. Lee, M. J. Assante, and T. Conway. “German Steel Mill Cyber Attack”. In: *Industrial Control Systems* (2014), pp. 1–15.

[100] A. Legay, B. Delahaye, and S. Bensalem. “Statistical Model Checking: An Overview”. In: *RV*. Vol. 6418. LNCS. Springer, 2010, pp. 122–135.

[101] M. Leucker and C. Schallhart. “A brief account of runtime verification”. In: *Journal of Logic Programming* 78.5 (2009), pp. 293–303.

[102] J. Ligatti, L. Bauer, and D. Walker. "Edit automata: enforcement mechanisms for run-time security policies". In: *International Journal of Information Security* 4.1-2 (2005), pp. 2–16.

[103] J. Ligatti, L. Bauer, and D. Walker. "Run-Time Enforcement of Nonsafety Policies". In: *ACM Trans. Inf. Syst. Secur.* 12.3 (2009).

[104] Y. Liu, P. Ning, and M. K. Reiter. "False Data Injection Attacks against State Estimation in Electric Power Grids". In: CCS '09. Association for Computing Machinery, 2009, 21–32.

[105] Rocchetto M. and N. O. Tippenhauer. "On Attacker Models and Profiles for Cyber-Physical Systems". In: *ESORICS 2016*. Vol. 9879. LNCS. Springer, 2016, pp. 427–449.

[106] Ibtissem Ben Makhlouf and Stefan Kowalewski. "An Evaluation of Two Recent Reachability Analysis Tools for Hybrid Systems". In: *Analysis and Design of Hybrid Systems 2006*. Elsevier, 2006, pp. 377–382.

[107] Z. Manna and A. Pnueli. *A Hierarchy of Temporal Properties*. Tech. rep. Stanford University, 1987.

[108] F. Martinelli and I. Matteucci. "Through Modeling to Synthesis of Security Automata". In: *ENTCS* 179 (2007), pp. 31–46.

[109] A. P. Mathur and N. O. Tippenhauer. "SWaT: a water treatment testbed for research and training on ICS security". In: *2016 International Workshop on Cyber-physical Systems for Smart Water Networks (CySWater)*. 2016, pp. 31–36.

[110] MATLAB. *9.7.0.1190202 (R2019b)*. Natick, Massachusetts: The MathWorks Inc., 2018.

[111] S. E. McLaughlin. "CPS: stateful policy enforcement for control system device usage". In: *ACSAC*. ACM, 2013, pp. 109–118.

[112] MIKE McQuade. *The untold story of NotPetya, the most devastating cyberattack in history*. 2018.

[113] J. Milošević, H. Sandberg, and K. H. Johansson. "Estimating the Impact of Cyber-Attack Strategies for Stochastic Networked Control Systems". In: *IEEE Trans. Control. Netw. Syst.* 7.2 (2020), pp. 747–757.

[114] S. Mohan et al. "S3A: secure system simplex architecture for enhanced security and robustness of cyber-physical systems". In: *HiCoNS*. ACM, 2013, pp. 65–74.

[115] Fereidoun Moradi et al. "Building Attack Models for Security Analysis of CPS". In: (2020).

[116] NERC-CIP. *Critical Infrastructure Protection*. North American Electric Reliability Corporation, 2008. URL: http://www.nerc.com/cip.html.

[117] V. Nigam, C. Talcott, and A. A. Urquiza. "Towards the Automated Verification of Cyber-Physical Security Protocols: Bounding the Number of Timed Intruders". In: *ESORICS 2016*. Vol. 9879. LNCS. Springer, 2016, pp. 450–470.

[118] G. Nikolakopoulos and S. Manesis. *Introduction to Industrial Automation*. Taylor & Francis Group, 2018.

[119] US NIST. "Guidelines for smart grid cyber security (vol. 1 to 3)". In: *NIST IR-7628, Aug* (2010).

[120] M. Okamoto. "Some inequalities relating to the partial sum of binomial probabilities". In: *Annals of the institute of Statistical Mathematics* 10.1 (1959), pp. 29–35.

[121] P. C. Ölveczky and J. Meseguer. "Semantics and pragmatics of Real-Time Maude". In: *Higher-Order and Symbolic Computation* 20.1-2 (2007), pp. 161–196.

[122] Shiva P., Zhen N., and Naresh M. "Real-time cyber physical system testbed for power system security and control". In: *International Journal of Electrical Power Energy Systems* 90 (2017), pp. 124–133.

[123] Ron J. Patton, Paul M. Frank, and Robert N. Clarke. *Fault Diagnosis in Dynamic Systems: Theory and Application*. USA: Prentice-Hall, Inc., 1989.

[124] H. Pearce et al. "Smart I/O Modules for Mitigating Cyber-Physical Attacks on Industrial Control Systems". In: *IEEE Transactions on Industrial Informatics* 16.7 (2020), pp. 4659–4669.

[125] G. Pedroza, L. Apvrille, and D. Knorreck. "AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties". In: *NOTERE*. IEEE, 2011, pp. 1–10.

[126] P. S. Pinisetty S.and Roop et al. "Runtime Enforcement of Cyber-Physical Systems". In: *ACM TECS* 16.5s (2017), 178:1–178:25.

[127] S. Pinisetty et al. "Runtime Enforcement of Regular Timed Properties". In: SAC 2014. 2014, pp. 1279–1286.

[128] B. Radvanovsky. "Project shine: 1,000,000 internet-connected SCADA and ICS stystems and counting". Tofino Security. 2013.

[129] R. Rajkumar et al. "Cyber-physical systems: the next computing revolution". In: *DAC*. ACM, 2010, pp. 731–736.

[130] P. J. Ramadge and W. M. Wonham. "Supervisory Control of a Class of Discrete Event Processes". In: *SIAM J. Control Optim.* 25.1 (1987), 206—230.

[131] Raspberry Pi. *Raspberry Pi 4 Model B*. 2019. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.

[132] S. Ratschan and Z. She. "Safety verification of hybrid systems by constraint propagation-based abstraction refinement". In: *ACM Transactions on Embedded Computing Systems* 6.1 (2007), p. 8.

[133] M Rocchetto and N. O. Tippenhauer. "CPDY: Extending the Dolev-Yao Attacker with Physical-Layer Interactions". In: *ICFEM 2016*. Ed. by Kazuhiro Ogata, Mark Lawford, and Shaoying Liu. Vol. 10009. LNCS. 2016, pp. 175–192.

[134] N. Roohi. "Remedies for building reliable cyber-physical systems". PhD thesis. University of Illinois at Urbana-Champaign, 2017.

[135] F. B. Schneider. "Enforceable security policies". In: *ACM Transactions on Information and System Security* 3.1 (2000), pp. 30–50.

[136] J. Slay and M. Miller. "Lessons Learned from the Maroochy Water Breach". In: *Critical Infrastructure Protection*. Vol. 253. IFIP. Springer, 2007, pp. 73–82.

[137] R. Spenneberg, M. Brüggerman, and H. Schwartke. "PLC-Blaster: A Worm Living Solely in the PLC". In: *Black Hat*. 2016, pp. 1–16.

[138] K. A. Stouffer, J. A. Falco, and K. A. Scarfone. *Guide to industrial control systems (ICS) security: Supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and other control system configurations such as programmable logic controllers (PLC)*. 2011.

[139] R. Taormina et al. "A toolbox for assessing the impacts of cyber-physical attacks on water distribution systems". In: *Environmental Modelling Software* 112 (2019), pp. 46–51.

[140] Positive Technologies. *Cybersecurity threatscape –Q2 2018*. 2018.

[141] A. Teixeira et al. "A secure control framework for resource-limited adversaries". In: *Automatica* 51 (2015), pp. 135–148.

[142] D. Thomas and P. Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[143] D. I. Urbina et al. "Limiting the Impact of Stealthy Attacks on Industrial Control Systems". In: *ACM CCS*. ACM, 2016, pp. 1092 –1105.

[144] A. Valdes and S. Cheung. "Communication pattern anomaly detection in process control systems". In: *2009 IEEE Conference on Technologies for Homeland Security*. 2009, pp. 22–29.

[145] R. Vigo. "The Cyber-Physical Attacker". In: *SAFECOMP 2012*. Vol. 7613. LNCS. Springer, 2012, pp. 347–356.

[146] Roberto Vigo. "Availability by Design: A Complementary Approach to Denial-of-Service". PhD thesis. Danish Technical University, 2015.

[147] V. Vladimerou et al. "STORMED Hybrid Systems". In: *ICALP 2008*. Ed. by Luca Aceto et al. Vol. 5126. LNCS. Springer, 2008, pp. 136–147. DOI: 10.1007/978-3-540-70583-3.

[148] Y. Wang et al. "SRID: State Relation Based Intrusion Detection for False Data Injection Attacks in SCADA". In: *Computer Security - ESORICS 2014*. Springer International Publishing, 2014, pp. 401–418.

[149] D. C. Wardell et al. "A Method for Revealing and Addressing Security Vulnerabilities in Cyber-physical Systems by Modeling Malicious Agent Interactions with Formal Verification". In: *Procedia Com. Sc.* 95 (2016), pp. 24 –31.

[150] W. Wolf. *FPGA-based system design*. Pearson education, 2004.

[151] K. C. Zeng et al. "All your {GPS} are belong to us: Towards stealthy manipulation of road navigation systems". In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, pp. 1527–1544.

[152] B. Zhu, A. Joseph, and S. Sastry. "A Taxonomy of Cyber Attacks on SCADA Systems". In: *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*. 2011, pp. 380–388.

[153] Q. Zhu and T. Basar. "Game-theoretic methods for robustness, security, and resilience of cyberphysical control systems: games-in-games principle for optimal cross-layer resilient control systems". In: *IEEE Control Systems* 35.1 (2015), pp. 46–65.

[154] N. Zilberman et al. “NetFPGA: Rapid prototyping of networking devices in open source”. In: *ACM SIGCOMM Comput. Commun. Rev.* 45.4 (2015), pp. 363–364.

[155] A. Zimba et al. “Crypto mining attacks in information systems: An emerging threat to cyber security”. In: *Journal of Computer Information Systems* 60.4 (2020), pp. 297–308.