Discrete Control of Response for Cybersecurity in Industrial Control

Gwenaël Delaval, Ayan Hore, Stéphane Mocanu, Lucie Muller, Eric Rutten

Discrete Control of Response for Cybersecurity in Industrial Control

Gwenaël Delaval, **Ayan Hore**, **Stéphane Mocanu**, **Lucie Muller**, **Éric Rutten**

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

Abstract: Cybersecurity in Industrial Control Systems (ICS) is a crucial problem, as recent history has shown. A notable characteristic of ICS, compared to Information Technology, is the necessity to take into account the physical process, and its specific dynamics and effects on the environment, when considering cybersecurity issues. Intrusion Detection Systems have been studied extensively. In our work, we address the less classic topic of response mechanisms, and their automation in a self-protection feedback loop. More precisely, we address self-protection seen as resilience, where the functionality of the system is maintained under attacks, be it in a degraded mode. We model this as a Discrete Event Systems supervisory control problem, involving a model of the plant's possible behaviors, a model of considered attacks, and a formulation of the control objectives. We consider a case study, and perform a prototype implementation and simulation, using the Heptagon/BZR programming language and compiler/code generator, and targeting a multi-PLC experimental platform.

Keywords: Industrial Control, Cybersecurity, Discrete Event Systems, Supervisory Control

1. INTRODUCTION

1.1 Industrial Control Systems and Cybersecurity

Modern Industrial Control Systems (ICS) are communicating devices (software or electronic components but also mechanical or hydraulic) which interact in order to achieve the control objective of a physical process (manufacturing, chemical or energy systems). They are often considered a critical infrastructure in the sense that their failure may lead to severe damages to the environment and fatalities. Traditionally ICS communication relies on proprietary networks and protocols. In the last twenty years TCP/IP and Internet related technology become more and more present due to their low cost and interoperability compared to legacy solutions. Opening to these technologies comes with a risk, because when ICS are open to remote maintenance, engineering and operations management, they are also open to internet threats. In the last decade, events and exploits (Stuxnet [Langner [2013]], BlackEnergy, Industroyer, WannaCry) demonstrated that cyber attacks may lead to physical damage on the process and involve losses of millions of euros. The consequences of the last in date major threat, LockerGoga, that infected Norsk Hydro on May 2019 where evaluated between 56 and 66 M€ [Hydro [2019]].

Due to the fact that ICS are used to control a physical process, the security deployment presents differences w.r.t. Information Technology (IT) systems. An extensive analysis is provided in the classical NIST guide [Stouffer et al. [2015]]. Relevant to our study are that firstly, in opposition with classical IT requirements analysis based on the CIA triad (Confidentiality, Integrity, Availability in that order of importance), the most important cybersecurity requirement for ICS is system availability, because losing process control is the worst case event. Integrity is the second requirement in order, because tampering control data may lead to process damage. Confidentiality is the less important requirement as data leaks may lead to some financial or reputation damage but unlikely to casualties.

Turning back to availability which is the paramount requirement of ICS system that will mean from the point of view of cybersecurity that classical solutions like rebooting or reloading the operation system in case of attack or virus infection cannot be applied. Shutting down an industrial facility is a long process that has to respect a strict procedure and it is common for critical ICS to be maintained operational even under attack. It follows that the operational characteristic of interest is the *resilience*, i.e., the capacity of the system to maintain a minimal quality of service while under attack.

One way to achieve resilience is to increase redundancy of the critical assets as in the case of fault tolerant systems. While this classical approach in safety is proved in case of failure, it may not be adequate in case of attack. Indeed while dependability and fault tolerance are designed in order to handle a set of well characterized accidental failures, they are not designed to handle intentional attacks. Cybersecurity approaches cannot consider only known scenarios but also the possibility of new form of system malfunction due to malevolent attacks (the so called "0-day" vulnerability exploitation). In case of detection of "strange" behaviour and doubtful networks activity the security management has to react in order to keep a minimal *safe* level of system function.

1.2 Response to attacks and its automation

Typically the response to attacks is triggered by a Security Information and Event Management (SIEM) system upon the reception and analysis of a security event sent by an Intrusion Detection Systems (IDS). Intrusion Detection research has been an intense activity field in IT in the last forty years. A classification and basic definitions may be found in the classical paper Debar et al. [1999]. From the point of view of detection method IDS may use a model of the normal behaviour to detect *anomalies* or use a pattern of the attack: those are the so called *signature-based* IDS. Various approaches are considering different models for detection. For instance Carcano et al. [2011] are building a stochastic model of the cyberphysical system for a probabilistic detection while Koucham et al. [2016] are using an approach based on security patterns learnt by property mining. Once notifications and alarms are being produced by such IDS, they can be received by the SIEM, where the operators of ICSs can analyse them and take appropriate counter measures, in a process that is considered manual, and not part of the research topic.

A less studied research topic is to consider specifically the response to attacks on ICS, and its automation, by analysing what are the classes of attacks that can potentially occur, what are the available means to use as counter-measures, and what are the policies and criteria for deciding on the most appropriate counter-measures. More precisely, depending on the nature of the attack, as well as the current state of the system and the available protection actions, there is a decision problem that has to be solved in a feedback loop enforcing self-protection. Only few research has been conducted on this field because the reaction mechanism is very dependent on the application field. A self-protection approach for smart-grids is presented in Kabir-Querrec et al. [2015]. Other approaches are based on the use of a backup system as in Babay et al. [2019] or a robust estimation algorithm able to correct corrupted input data as in Zhu [2014].

1.3 Contributions of this work

Our approach is focusing on a class of response mechanisms by isolation of attacked components and reconfiguration of the ICS in order to keep it under safe control. We present the following contributions : (i) our approach to self-protection seen as resilience ; (ii) a generic model as a Discrete Event System (DES) supervisory control problem, encoded in the Heptagon/BZR language ; (iii) a case study, and a prototype implementation and simulation.

2. DISCRETE CONTROL WITH HEPTAGON/BZR

Initially defined in the framework of language theory by Ramadge and Wonham [1989], the supervisory control of DES, and the algorithmic tool for Discrete Controller Synthesis (DCS) has been adapted to symbolic Labelled Transition Systems (LTS) representing finite state automata, and implemented in tools within the reactive languages by Marchand and Samaan [2000] and Berthier and Marchand [2014]. DCS is applied on an LTS representing possible behaviors of a system, its variables being partitioned into controllable ones (c) and uncontrollable ones (u). For a given control objective (e.g., staying invariantly inside a given subset of states, considered "good"), the DCS algorithm automatically computes, by exploration of the state space, the constraint on controllable variables, depending on the current state, for any value of the uncontrollables, so that remaining behaviors satisfy the objective. This constraint is inhibiting the minimum possible behaviors, therefore it is called *maximally permissive*. If no solution is found, then DCS plays the role of verification. The computational complexity w.r.t. using DCS is of course an issue, in FSM as well as in Petri nets based supervisory control techniques [Yang and Hu [2019]] because they involve unfolding the marking graph, but on the one hand there are ongoing research works dealing with modular DCS, and on the other hand the size of there are methods to decompose systems into subsystems (see Section 4.6).

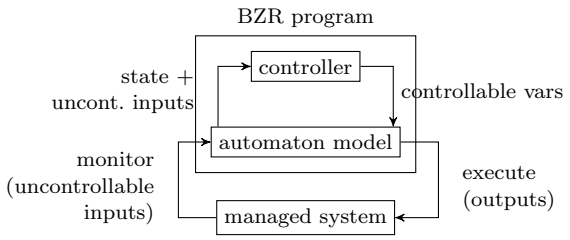

Fig. 1. Heptagon/BZR: the `contract` is transformed by compilation and DCS into a controller.

In our work, we use the Heptagon/BZR programming language and compiler to build automata models and apply DCS, as shown in Fig. 1. Using this language enables us to constructs models which are at the same time formal, for the DCS tool, and executable, for our application, without any loss in generality. Heptagon/BZR [Delaval et al. [2013]] is a synchronous programming language which allows to model systems in term of parallel composition of reactive finite state automata. A program written in Heptagon/BZR is composed of *nodes*, each node being defined by a set of *inputs* (each input modelling a stream of inputs, typically coming from the environment), a set of *outputs* (e.g., streams of actions towards the environment), and a set of equations and/or automata defining the relation between the sequences of inputs and outputs.

For example, the node in Fig. 2 describes the simplified model of a program running on a Programmable Logic Controller (PLC). This program can be in three different states: `Nominal`, `Degraded` and `Safe` (the subsystem controlled by this program has been disconnected). This node is defined with two input streams: `deg` (resp. `safe`) is a Boolean input which forces the program to go to its degraded (resp. safe) mode. The output `e_stop` emits an "emergency stop" event towards the subsystem when the program is in safe mode.

The table below shows an execution example of this node. At each execution step, the values of the two inputs are read, the current state and the outputs are computed from the previous state and the current values of inputs.

`deg`	0	0	0	1	0	0	0
`safe`	0	0	0	0	0	1	0
State	N	N	N	D	D	S	S
`e_stop`	0	0	0	0	0	1	1

```
node prog(deg,safe:bool) = (e_stop:bool)
let
  automaton
    state Nominal
      do e_stop = false
      unless deg then Degraded
           | safe then Safe
    state Degraded
      do e_stop = false
      unless safe then Safe
    state Safe
      do e_stop = true
  end
tel
```

Fig. 2. Heptagon/BZR node example: model of program

The Heptagon/BZR language allow then to express parallel composition of automata: e.g., we can instantiate twice the `prog` node with distinct inputs and outputs:

```
e_stop1 = prog(deg1, safe1);
e_stop2 = prog(deg2, safe2);
```

Then, our language allows to express *contracts* on nodes, so as to define temporal properties on inputs and outputs. For example, we can express that at each instant, if the first program is in safe mode, then the second one must be also in safe mode: `enforce (e_stop1 => e_stop2)`. These temporal properties will be *enforced* by a *controller*, automatically generated by a *DCS tool* from an equational representation of the program. This equational representation is obtained directly by compilation of the Heptagon/BZR program: a program can then be seen as well as a computable model. This computed controller will act, at execution time, on *controllable variables* such that the property to be enforced will be satisfied, at each execution step, whatever be the future values of inputs, depending on the current state (resp. c, u and s in Fig. 1).

In the node given on Fig. 3, we consider that the synthesized controller can act on the variables `safe1` and `safe2`: i.e., that the controller can force the two entities to go in degraded mode. The aforementioned property is enforced by this controller, without being explicitly programmed.

3. ICS AND CYBER-SECURITY PROBLEM

According to ANSSI [1] and NIST [2] classification, both based on CIM [3], model entities in an ICS are classified from the point of view of their role in the ICS: sensors/actuators, controllers and operator consoles, Supervisory Control and Data Analysis (SCADA) servers and clients, enterprise resource planning (ERP) and data analysis.

In our approach we consider the reaction for the self-protection of the critical part of the ICS. The self-reconfiguration will concern the controllers which are targets of the attacks although the intermediate steps of the attacks may corrupt SCADA and ERP devices.

[1] Agence nationale de la sécurité des systèmes d'information — French Cybersecurity authority
[2] National Institute of Standards and Technology
[3] Computer Integrated Manufacturing

```
node two_progs(deg1,deg2:bool)
            = (e_stop1,e_stop2:bool)
contract
  enforce (e_stop1 => e_stop2)
  with (safe1, safe2:bool)
let
  e_stop1 = prog(deg1, safe1);
  e_stop2 = prog(deg2, safe2);
tel
```

Fig. 3. Heptagon/BZR node example: controlled programs

The design of the reaction mechanism is stated as a DES supervisory control problem, using our reactive programming language Heptagon/BZR. Other approaches in the state of the art of DES for security and privacy have been considering obfuscation as e.g., Jacob et al. [2016], Ji et al. [2018]. Differently to these approaches, we consider the reaction to attacks from a resilience perspective.

3.1 Response and resilience

We consider attack scenarios that will compromise controllers such as PLC by, for example, *Denial of Service* when the control function is no more computed by the PLC, a *Variable Tampering* when the data in the PLC memory is compromised or a *Program Tampering* when the control program is modified.

In our approach, to improve the resilience of the ICS in case of cyber attacks, we propose a self-reconfiguration reaction at two levels: (i) network level: we reconfigure interconnection devices so that the compromised PLC is isolated; (ii) controllers level: we reconfigure the programs running in the available PLCs so that the control function of the compromised one are taken over by another.

While the first part of the reaction (network level) does not need any special features of the architecture (standard manageable interconnection devices are sufficient), redeployment of control functions is possible only if the sensors and actuators are accessible to several PLC over the network. In the next section we specify the characteristics of the ICS architecture supporting resilience.

3.2 Overall ICS architecture

We focus on the lower levels (actuators, sensors and controllers) of an ICS architecture. We assume:

- the physical process may be technologically divided in **subsystems**, corresponding to either control loops or components of the plant (tanks, mixers, heaters, etc);
- sensor and actuators are all connected to **RTU (Remote Terminal Units)** which communicate through an Ethernet network. The RTU's embedded programs only support an emergency stop (E-STOP) function, that can be remotely triggered, to put the local loop in safe mode;
- controllers are **PLC**s, and may differ in terms of memory and performance e.g., processor throughput;
- the **local area network (LAN)** is built around manageable Ethernet switches, with ports that may be shut down for disconnection from the network.

Figure 4 shows an example ICS with the previous characteristics, with two subsystems, three RTUs and two PLCs.

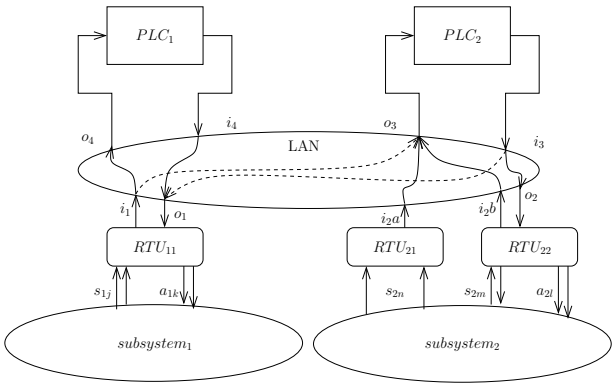

Fig. 4. An example of considered ICS architecture.

3.3 Response to attacks in a reaction feedback loop

As shown in the feedback loop of Figure 5, the reaction is triggered by the *alarms* raised by the detection block (of which the detail is out of our scope): we assume that several IDS alerts are correlated e.g., within a SIEM in order to detect and identify attacks. We consider that a knowledge base of possible actions is available and, depending on the alarms and the *states* of the PLCs and programs cw_i (critical phase) and sw_i (switchable phase), as well as a manual order to go to emergency mode e_i, the self-protection manager decides on the reaction: (i) isolate the attacked component from the rest of the ICS (by disconnecting communications); (ii) reorganize the functionalities required for the ICS to run properly, i.e., reorganize control programs so that safety as well as computing infrastructure capacity requirements are met.

The reaction manager will deliver a response in the form of a stream of values to the ICS concerning: the isolation of the attacked PLC (which is handled by the switches), the versions of the programs to be activated mode_i, as well as its location amongst PLCs $\mathrm{ex\_loc}_i$, according to the constraints to be satisfied by the global system.

3.4 Physical and hardware related constraints

Due to the critical mission of most of the ICSs, reconfiguration actions cannot take place at any moment. For instance, during some technological steps it might be impossible to change the controller or the program version. If for example, the duration of an operation is controlled by an internal timer block, then program migration will

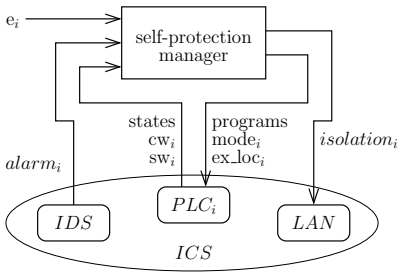

Fig. 5. Self-protection feedback loop in response to attacks.

be impossible if the exact step duration has to be respected. Also, PLCs are typically resource constrained and the control programs must satisfy explicit response time bounds, so that reallocating a control program from a PLC to another might jeopardize real-time performance. In order to take into account such physical and computing resource constraints, we make assumptions and impose explicit constraints to be taken into account.

Control program assumptions and constraints. Programs are written in IEC 61131 languages like SFC or Ladder Diagrams, under several versions, distinguished by:

QoS: quality of service of the computed control: a more elaborate version can provide with better precision, or performance, e.g., involving complex optimization;
execution time: duration of the program version within each PLC cycle, can differ w.r.t. complexity;
size in the memory: the code of each program version occupies part of the PLC's bounded memory.

Typically, we consider the following versions:

nominal delivering full functionality, with best quality, e.g., precision of command, or optimization ;
degraded delivering limited functionality, e.g., sub-optimal control, with a smaller use of resources;
safe mode or emergency stop where functionality is not delivered any more, but the essential controls are kept on the physical subsystem, in a safe state.

A subsystem can be ordered to go to emergency stop at any time, by the human operator of the ICS or by the self-protection manager. We consider that a program P_i can be in a critical phase, and can have phases where no version switching is allowed. This information is made available to the self-protection manager through Boolean variables resp. cw_i and sw_i. At runtime, at each execution cycle of the PLC, only some versions are computed by submitting their execution to a condition excluding the others.

PLC and RTU related assumptions. Each PLC is characterized, for the purposes of this work, by:

- the size of the memory where programs are loaded statically, before the starting of the ICS;
- the cycle time, within which PLCs must read data from the various inputs, execute codes and then write the outputs, along with communication overheads. This defines a maximum duration bounding the execution of active programs within one cycle;
- the manufacturer or type, as not all of them are interoperable w.r.t. execution of control programs.

We assume that not all PLCs are attacked at the same time, i.e., for n PLCs, only $n-1$ out of n can be. We also assume that, thanks to their simplicity and limited interactivity, RTUs will not be targeted by attacks.

Communication network assumption. A switched LAN is interconnecting the PLC and RTU's. We assume that switches are manageable and ports can be shut down if needed in order to isolate compromised devices. We do not consider attacks targeting the network as they are not specific to ICS. We consider that general IT countermeasures were deployed and switches are secured.

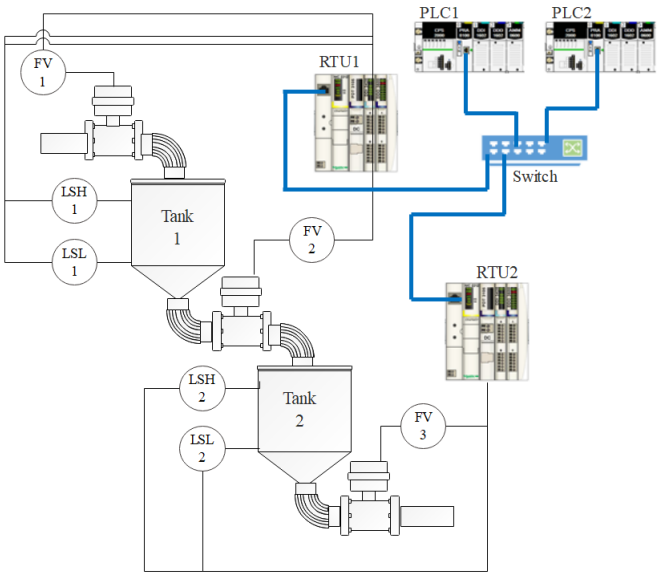

Fig. 6. The simple use-case of two cascading tanks.

3.5 Interactions between control programs

static priorities can designate programs to be more or less critical: higher priority programs should be executed in better version than lower priority ones.

dynamic priorities, e.g., depending on a state information from the physical plant or a program

dependencies between emergency stop modes: when some subsystem is in emergency stop mode then some others should be too; but not necessarily reciprocally, e.g., if P1 in safe mode then P2 and P3 as well (but not P1 if P2 or P3 in safe mode).

3.6 Control informal specification

To summarize, the following policies must be enforced by the controller of the re-configurations of the ICS :

(i) every subsystem should be under the control of at least one and at most one version of a control program, executed on some PLC or RTU;
(ii) a control program can be executed on a PLC only if it has a version which is compatible with the PLC type and available in the memory of that PLC;
(iii) the total code of control programs available on a PLC must not exceed its memory capacity;
(iv) the duration of the execution of all programs active in the same cycle must be less than the Cycle Time;
(v) static priorities between programs must hold;
(vi) dynamic priorities between programs must hold;
(vii) subsystems can be put in emergency stop at any time;
(viii) dependencies of emergency stop modes must hold;
(ix) a PLC under attack should not execute any program.

The above logical properties might be satisfied by several different configurations. The choice amongst them must be made according to other criteria, for example more qualitative, like cost in resources or quality and performance, upon which optimization can be done. In our case, we will want to maximize Quality of Service (QoS), i.e., maximize number of control programs in nominal mode.

3.7 Use case scenario

We defined a very simple use-case scenario to test the technical feasibility of the actions and illustrate the concepts. It corresponds to the class of architectures of ICS shown earlier in Fig. 4. The process consists of two cascading tanks as in Fig. 6. The transformation process consists of two technological steps: a first transformation process takes place in Tank1 then the intermediate product is transferred to Tank2 for a final transformation. Three actuators (valves FV1, FV2 and FV3) are used to fill/empty the tanks. The state of the tank (full/empty) is detected by the level sensors LSH1/LSL1 resp. LSH2/LSL2.

Two RTUs are used to manage the sensor and actuators: one for LSH1, LSL1, FV1, FV2 and a second one for LSH2, LSL2 and FV3. A PLC is used to control the Tank 1 and the transfer to Tank2. A second PLC is used to control Tank2. The state of the Tank 2 is read periodically by PLC1 from PLC2. All devices are connected by a switch.

The process is simulated but it interacts with real RTU and PLC devices through our hardware-in-the-loop ICS sandbox [Mocanu et al. [2019]]. Both PLCs contain two versions of the control program: one in nominal mode when both PLCs are available and a second version for the case when one of the PLC is attacked. The network actions (shutting down Ethernet ports on the switch) are implemented in Expect scripts [4] . Program switching into the PLC is actually a branching in the main program which is activating on an external signal.

4. DES MODELLING & SUPERVISORY CONTROL

In this section, we show how the system considered can be defined as a DES model, where each element (i.e., PLCs and control programs) is modelled as an automaton, with equations on states defining the dynamic properties of these elements (duration cycle, execution localization of programs, availability of PLCs). Then the security properties defined as DCS objectives, to be enforced online by a controller, itself synthesized offline.

4.1 Problem Definition and Overall Model Architecture

The problem stated in previous sections is defined as:

- a set of n control programs $P_i, i = 1, \ldots, n$;
- a set of p PLCs $C_j, j = 1, \ldots, p$;
- $\max_j$ is the maximum cycle duration of PLC C_j;
- n_{ij} is the duration of the *nominal* version of program P_i on PLC C_j;
- d_{ij} is the duration of the *degraded* version of program P_i on PLC C_j.

The overall model is the parallel composition of all automata representing PLCs and programs, defined in a Heptagon/BZR node. The Boolean inputs of this main node are (cf. Section 3.4, and Fig.5):

- $\texttt{alarm}_i, i = 1, \ldots, p$, notifying alarms on PLC C_i;
- $\texttt{cw}_i, i = 1, \ldots, n$, `true` when P_i in critical section;
- $\texttt{sw}_i, i = 1, \ldots, n$, `true` when P_i is "switchable";
- $\texttt{es}_i, i = 1, \ldots, n$, triggering emergency stop of P_i.

[4] https://core.tcl-lang.org/expect/home

Outputs of the main node, towards the environment, are:

- $\texttt{mode}_i$: the version of program P_i to be executed (within values `Nominal`, `Degraded` and `Safe`);
- $\texttt{ex\_loc}_i$: the execution location of the program P_i (within values $\texttt{PLC}_1, \ldots, \texttt{PLC}_p$).

A *contract* will be defined on this main node, in order to synthesize a controller, which will enforce *objectives* at execution, using controllable features: the version and localization of programs. The *controllable variables* defined in this node, i.e., the variables on which the synthesized controller will act, are then: $\texttt{cd}_i$ and $\texttt{cs}_i$, Boolean variables allowing the controller to force the program P_i to go in `Degraded` (resp. `Safe`) version, and $\texttt{el}_i$, within values $\texttt{PLC}_1, \ldots, \texttt{PLC}_p$, allowing the controller to chose the PLC on which the program P_i has to be executed.

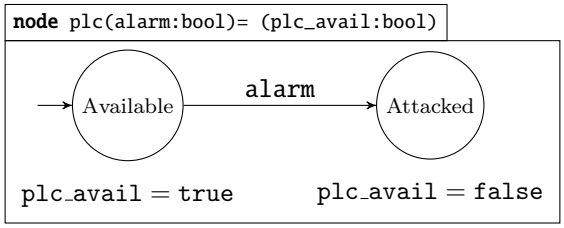

Fig. 7. PLC Heptagon/BZR model

4.2 Behavioral models

PLC models We represent a PLC as a two-state automaton (see Fig. 7). The node `plc` takes as input the `alarm` stream, signalling that an attack has been detected on this PLC. The output `plc_avail` is true whenever the PLC is considered to be available; and becomes and remains false once an attack has been detected. `plc` nodes are instantiated for all PLCs and composed in parallel:

$$\texttt{plc\_avail}_1 = \texttt{plc}(\texttt{alarm}_1);$$
$$\vdots$$
$$\texttt{plc\_avail}_p = \texttt{plc}(\texttt{alarm}_p);$$

Additionally, the main contract comprises an *environment hypothesis* on alarms: there will be, at any execution step, at least one PLC available. This is expressed on $\texttt{plc\_avail}_i$ variables as: $\textbf{assume} \bigvee_{i=1}^{p} \texttt{plc\_avail}_i$.

Control program models A program is modelled with the automaton given in Fig. 8. $\texttt{dn}_i$ and $\texttt{dd}_i$ are *static* parameters of the node `prog`, and are the duration values of the program on PLC C_i, in nominal or degraded mode.

The outputs of this node are `mode`, `exec_loc` and $\texttt{dur}_i$. `mode` is the current version executed by the program (`Nominal`, `Degraded`, or special `Safe` value for emergency stops). `exec_loc` is the PLC on which the program has to be executed at each step. For each PLC, $\texttt{dur}_i$ is the current execution duration of the program on the PLC C_i (0 if the program is not executed on this PLC).

The variable `c_exec_loc` is meant to be controllable at main level; it allows the controller to define the PLC on which the program should be executed. This request from the controller is effective in Degraded mode (equation

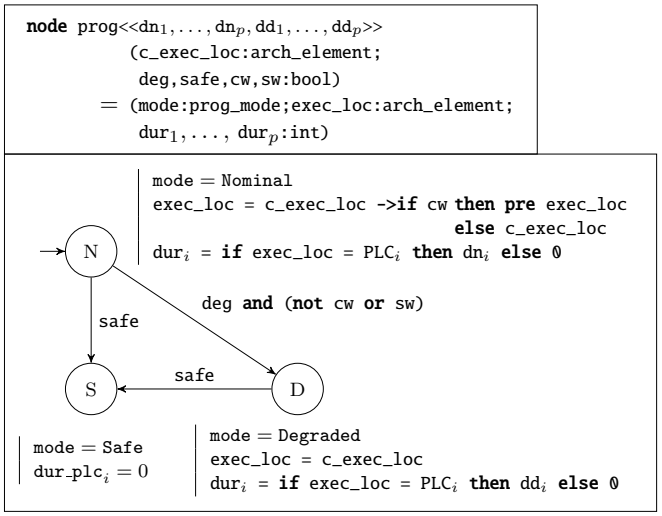

Fig. 8. Program Heptagon/BZR model

$\texttt{exec\_loc} = \texttt{c\_exec\_loc}$), and in Nominal mode, if the program is not in "critical wait" section (i.e., when the input `cw` is true); otherwise the execution location will be the previous one (`pre` $\texttt{exec\_loc}$). In Nominal mode, the value of `exec_loc` is initialized at first step with the first value of `c_exec_loc` (operation `->`).

Variables `deg` and `safe` are also meant to be controllable: they allow the controller to enforce the current mode of the program. The transitions show that not every mode can be taken at every instant: whereas the Safe mode is always reachable (as it is the "emergency stop"), a program can only switch from Nominal to Degraded if either it is not in a critical section (`cw` is false), or it is in a critical section but has been recognized as "switchable" (`sw` is true).

This node and its automaton make the system satisfying by construction the properties (i) (as the output `mode` is of enumerated type, of values taken from the set $\{\texttt{Nominal}, \texttt{Degraded}, \texttt{Safe}\}$) and (vii), by the effect of the transitions from every state to the state `Safe`, and the instantiation given below of the node input `safe`.

This `prog` node is then instantiated for each program, in a parallel composition:

$$(\texttt{mode}_1, \texttt{ex\_loc}_1, \texttt{dur}_{11}, \ldots, \texttt{dur}_{1p}) = \texttt{prog}\langle\langle n_{11}, \ldots, n_{1p}, d_{11}, \ldots, d_{1p}\rangle\rangle (\texttt{el}_1, \texttt{cd}_1, \texttt{es}_1 \textbf{ or } \texttt{cs}_1, \texttt{cw}_1, \texttt{sw}_1);$$
$$\vdots$$
$$(\texttt{mode}_n, \texttt{ex\_loc}_n, \texttt{dur}_{n1}, \ldots, \texttt{dur}_{np}) = \texttt{prog}\langle\langle n_{n1}, \ldots, n_{np}, d_{n1}, \ldots, d_{np}\rangle\rangle (\texttt{el}_n, \texttt{cd}_n, \texttt{es}_n \textbf{ or } \texttt{cs}_n, \texttt{cw}_n, \texttt{sw}_n);$$

We recall that for each instantiation, n_{ij} (resp. d_{ij}) is the duration cost of program P_i on PLC C_j in nominal (resp. degraded) mode. We can here represent the fact that a version of a given program P_i does not exist on a PLC C_j by stating n_{ij} or $d_{ij} = \infty$.

4.3 Global Cost Model and Control Objectives

In addition to parallel instances of `plc` and `prog` nodes (see above), the global model also comprises the computation of the total duration of programs on each PLC:

```
dur_plc_1 = dur_11 + ... + dur_n1
⋮
dur_plc_p = dur_1p + ... + dur_np
```

These equations state that the total duration on each PLC C_j is the sum of the duration of each program P_i on this PLC ($\texttt{dur}_{ij}$), taken from instances of node `prog`.

Then, these variables can be used to define control objectives to be enforced by the synthesized controller. In relation to the properties introduced in Section 3.6:

- property (iv) (duration of execution of program versions active on the same PLC should be less than the cycle time of this PLC) can be stated as: **enforce** $\bigwedge_{i=1}^{p} \texttt{dur\_plc}_i \leq \max_i$. This synthesis objective also enforce the property (ii) (presence of a compatible version on the PLC), by defining $n_{ij} = \infty$ (resp. $d_{ij} = \infty$) if the nominal (resp. degraded) version of P_i is not available on PLC C_j.
- The property (ix) (a PLC under attack should not be used to execute any program) is enforced by stating that on any PLC which became not available, the total duration cost have to be 0: **enforce** $\bigwedge_{i=1}^{p} \neg \texttt{plc\_avail}_i \Rightarrow (\texttt{dur\_plc}_i = 0)$.
- Property (viii) (dependencies between safe/emergency stops modes of programs P_i and P_j) can be stated as: **enforce** $(\texttt{mode}_i = \texttt{Safe}) \Rightarrow (\texttt{mode}_j = \texttt{Safe})$.

4.4 One-step Optimization and Priorities

The synthesized controller, computed by the synthesis tool, is *maximally permissive*, i.e., the constraint on the program is minimal and thus keeps all the possible behaviors such that the properties are satisfied.

Within this set of possible behaviors, one must be chosen at execution. The solution presented here consist in applying a one-step optimization phase, at the end of the controller synthesis process. The controller will then, among the set of solutions, take a solution which will maximize, in our case, the number of programs in nominal mode.

To be able to perform this optimization phase, we add the following equations to the main node:

```
count_1 = if mode_1 = Nominal then 1 else 0;
⋮
count_n = if mode_n = Nominal then 1 else 0;
count = count_1 + ... + count_n
```

These equations define the variables $\texttt{count}_i$, equal to 1 if P_i is in nominal mode, 0 else; and `count` being the number of programs in nominal mode, at each execution step. Then, we can compute the controller enforcing each previously defined objective, and which will maximize the value of `count` at each step.

Static priorities between programs (property (v), i.e., preferring to keep highest priority programs in nominal modes), can be easily expressed by adding weights w_i on counts of programs nominal modes:

```
count = w_1 * count_1 + ... + w_n * count_n
```

For example, an absolute and total priority order $P_1 \prec \ldots \prec P_n$ (i.e., we want to prioritize nominal mode on P_n over any other one) can be defined with $w_i = n^i$.

Dynamic priorities (property (vi)) can be stated the same way, by adding the values w_i as inputs of the main node instead of as constants.

In summary, we described how a model can be built for an ICS architecture as specified informally in Section 3, and how control objectives can be formalized on the variables of the model for all required properties (i) to (ix) of Section 3.6, as well as the optimization objective. Hence, the self-protection as resilience problem has been formalized as a DES supervisory control problem, for which DCS tools can be used, and an executable controller can be obtained, to be integrated in the architecture.

4.5 Use case scenario

We show in this section the controller synthesis and simulation of a system corresponding to the use case of Section 3.7, made slightly more complex by the addition of one more program, for illustration purposes. It is composed of 3 programs (P_1 to P_3), running on 2 PLCs (PLC1 and PLC2). The duration cost of nominal and degraded versions are (in ms):

$$(n_{ij}) = \begin{pmatrix} 20 & 30 & 25 \\ 20 & 30 & 25 \end{pmatrix} \qquad (d_{ij}) = \begin{pmatrix} 10 & 15 & 15 \\ 10 & 15 & 15 \end{pmatrix},$$

with maximum durations $\max_1 = 50\,\text{ms}$, $\max_2 = 40\,\text{ms}$.

The Figure 9 shows a simulation of this system. Initially, all the programs are in nominal mode. P_1 and P_2 are placed on PLC1, and P_3 on PLC2 (as the total sum of durations exceed the maximum duration on PLC1). When an alarm is triggered on PLC1 (while P_1 is in a critical section), P_2 is executed in degraded mode on PLC2 with P_3, and P_1 is switched to safe mode (being in critical section, it cannot be switched to another PLC).

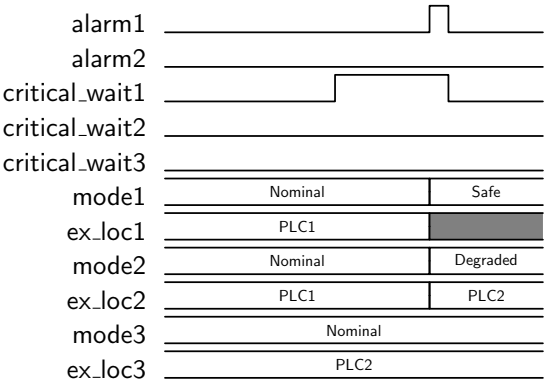

Fig. 9. Simulation of 3 programs running on 2 PLC

4.6 Method scalability

To evaluate the scalability of our method, we measure the synthesis time for models of systems of different sizes. The complexity of the synthesis being exponential in the size of the input model (size as number of inputs, states and controllable variables), the synthesis time is the bottleneck of our method. We take systems of size n, n being both

the number of programs and of PLCs. We consider that all programs can be executed on all PLCs, with $n_{ij} = 30\,\text{ms}$, $d_{ij} = 10\,\text{ms}$, and $\max_j = 50\,\text{ms}$.

Figure 10 shows the synthesis time for different values of n, with and without one-step optimization, performed on a 3.70 GHz CPU. This graph, besides showing the actual exponential costs of synthesis times, also points out an additional cost for one-step optimization, of several order of magnitudes compared with the synthesis cost without optimization. The execution time of the produced controller is itself linear. Note that this execution time does not have an impact on the real system as the computations are achived off-line.

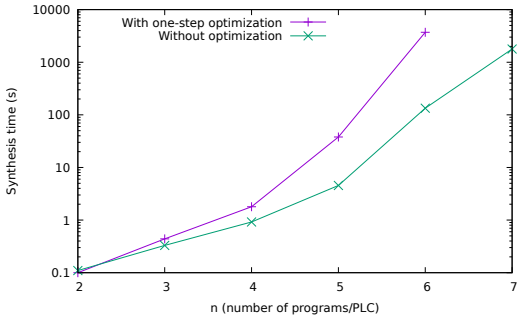

Fig. 10. Synthesis time for n programs and PLCs

We argue that, while our method is able to handle only systems of quite small sizes ($n \lesssim 7$), the combinatorial of solutions for systems of such sizes make them hard if not impossible to handle "manually", i.e., by programming without controller synthesis. However, these sizes are relevant to real-life industrial systems as even large size systems are using network segmented then can be decomposed in a set of smaller size systems. On sizes for which one-step optimization is not usable, simple optimizations with static and absolute priorities can be modelled with Heptagon/BZR, with the declaration order and values of controllable variables. Modelling bigger systems can take into account the system structure and benefit from modular synthesis with Heptagon/BZR (Delaval et al. [2014]).

5. CONCLUSIONS AND PERSPECTIVES

We presented an approach to the cybersecurity of ICS, with the automated reaction by self-protection to attacks, based on resilience mechanisms. Our contributions are in the proposal of a self-protection manager for reaction to attacks, its modeling and design involving Supervisory Control of DES and DCS tools, and treatment of a use case, using the Heptagon/BZR language.

Perspectives are in several directions: regarding the use of DCS and its complexity, and the scalability in size of ICSs, problems can be addressed by modularity, decomposing the problem so as to have hierarchical distributed controllers. On the side of cybersecurity automation and self-protection controllers, on the short term we will set up a larger size use-case experiment involving around 10 PLC and 20 RTU in order to be more realistic w.r.t. attacks that compromise communication between the self-protection manager and PLC, like Ethernet storms or Distributed Denial of Service. For the moment we assumed that the communication between the centralized self-protection manager and the non-compromised PLC is possible which may not always be the case. Distributing part of the decision manager on the PLC themselves will allow us to handle this issue.

REFERENCES

A. Babay, J. Schultz, T. Tantillo, S. Beckley, E. Jordan, K. Ruddell, K. Jordan, and Y. Amir. Deploying intrusion-tolerant scada for the power grid. In *2019 49th IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, June 2019.

N. Berthier and H. Marchand. Discrete Controller Synthesis for Infinite State Systems with ReaX. In *IEEE Int. Workshop on Discrete Event Systems WODES*, Cachan, France, May 2014.

A. Carcano, A. Coletta, M. Guglielmi, M. Masera, I. Nai Fovino, and A. Trombetta. A multidimensional critical state analysis for detecting intrusions in SCADA systems. *IEEE Trans. Industrial Informatics*, 7(2), 2011.

H. Debar, M. Dacier, and An. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805 – 822, 1999.

G. Delaval, É. Rutten, and H. Marchand. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, December 2013.

G. Delaval, S. Mak Karé Gueye, É. Rutten, and N. De Palma. Modular coordination of multiple autonomic managers. In *CBSE'14, Proc. of the 17th Int. ACM SIGSOFT Symposium on Component-Based Software Engineering, Lille, France, June 30 - July 4, 2014*, 2014.

Norsk Hydro. Cyber-attack on Hydro in brief, 2019. URL `https://www.hydro.com/en-NO/media/on-the-agenda/cyber-attack/`.

R. Jacob, J.-J. Lesage, and J.-M. Faure. Overview of discrete event systems opacity: Models, validation, and quantification. *Annual Reviews in Control*, 41, 2016.

Y. Ji, X. Yin, and S. Lafortune. Opacity enforcement by insertion functions under energy constraints. *IFAC-PapersOnLine*, 51(7): 291 – 297, 2018. 14th IFAC Workshop on Discrete Event Systems WODES 2018.

M. Kabir-Querrec, S. Mocanu, P. Bellemain, J.-M. Thiriet, and E. Savary. Corrupted GOOSE Detectors: Anomaly Detection in Power Utility Real-Time Ethernet Communications. In *GreHack*, Grenoble, France, nov. 2015.

O. Koucham, S. Mocanu, G. Hiet, J.-M. Thiriet, and F. Majorczyk. Detecting Process-Aware Attacks in Sequential Control Systems. In *NordSec*, Oulu, Finland, November 2016.

R. Langner. To kill a centrifuge, 2013. URL `https://www.langner.com/wp-content/uploads/2017/03/to-kill-a-centrifuge.pdf`.

H. Marchand and M. Samaan. Incremental design of a power transformer station controller using a controller synthesis methodology. *IEEE Trans. on Soft. Eng.*, 26(8):729 –741, 2000. ISSN 0098-5589.

S. Mocanu, M. Puys, and Thevenon P.-H. An Open-Source Hardware-In-The-Loop Virtualization System for Cybersecurity Studies of SCADA Systems. In *Journées C&ESAR Virtualisation et Cybersécurité*, Rennes, France, November 2019.

P.J. Ramadge and W.M. Wonham. On the supervisory control of discrete event systems. *Proc. IEEE*, 77(1), January 1989.

K. A. Stouffer, J. Falco, and K. Scarfone. Guide to industrial control systems (ICS) security, 2015.

B. Yang and H. Hu. Secure conflicts avoidance in multidomain environments: A distributed approach. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–12, 2019.

B. X. Zhu. *Resilient Control and Intrusion Detection for SCADA Systems*. PhD thesis, UC Berkeley, May 2014.