CRYSTAL Framework: Cybersecurity Assurance for Cyber-Physical Systems

(Technical Report)

Fereidoun Moradi[a,*], Sara Abbaspour Asadollah[a], Bahman Pourvatan[a], Zahra Moezkarimi[a], Marjan Sirjani[a]

[a]*School of Innovation, Design and Engineering, Mälardalen University, Västerås 722 20, Sweden*

Abstract

We propose CRYSTAL framework for automated cybersecurity assurance of cyber-physical systems (CPS) at design-time and runtime. We build attack models and apply formal verification to recognize potential attacks that may lead to security violations. We focus on both communication and computation in designing the attack models. We build a monitor to check and manage security at runtime, and use a reference model, called Tiny Digital Twin, in detecting attacks. The Tiny Digital Twin is an abstract behavioral model that is automatically derived from the state space generated by model checking during design-time. Using CRYSTAL, we are able to systematically model and check complex coordinated attacks. In this paper we discuss the applicability of CRYSTAL in security analysis and attack detection for different case studies, Pneumatic Control System (PCS), Temperature Control System (TCS), and Secure Water Treatment System (SWaT). We provide a detailed description of the framework and explain how it works in different cases.

Keywords: Cybersecurity, Cyber-Physical Systems (CPS), Formal verification, Model abstraction, Attack detection system, Runtime Monitoring, Tiny Digital Twin

*Corresponding author

Email addresses: fereidoun.moradi@mdu.se (Fereidoun Moradi), sara.abbaspour@mdu.se (Sara Abbaspour Asadollah), bahman.pourvatan@mdu.se (Bahman Pourvatan), zahra.moezkarimi@mdu.se (Zahra Moezkarimi), marjan.sirjani@mdu.se (Marjan Sirjani)

1. Introduction

Cybersecurity is considered as one of the highest priority targets in global policy and national security plans. There are increasingly challenging cybersecurity issues for governments and large companies in various sectors such as water supply system, energy production, transportation and smart machines. Because of rapid digitalization, a majority of manufacturing systems are no longer closed systems and are becoming systems with increasingly networked and cloud-based connectivity. The attack surface is hence expanded from known threats and known devices to additional security threats of new devices, protocols, and workflows.

Security assurance is a non-stop process. Companies need to continually assess their cybersecurity posture to ensure they are up to date with the latest security measures. We need to prepare our organizations and industrial companies by using proper tools, solutions and methodologies, both at the design phase and the operational phase of the system, and provide well-formed adaptation strategies to withstand failures. Formal methods provide an approach to verifying software systems, which can be particularly useful in the field of cybersecurity. By using formal methods, one can create a precise mathematical model of the system at design-time, which can be used to identify potential vulnerabilities, detect and diagnose flaws and errors, and verify that the system is secure and will behave as intended. Runtime verification and monitoring can also be used for resilience against cyberattacks by preventing and detecting cyberattacks and therefore can help in improving reaction time, reducing downtime, and ultimately saving money in the case of an attack.

Cyber Physical Systems (CPS) provide an outstanding foundation for digitalization and building advanced industrial systems and applications. CPSs are systems that integrate physical, computational, and communication subsystems. In a CPS, sensors are responsible for collecting data on the state of a physical process and submit them to the controllers. Controllers control the physical process using actuators. These systems are used in a wide variety of safety-critical applications, from automotive and avionic systems to robotic surgery and smart grids.

In this work, we give a thorough overview of the CRYSTAL[1] framework which can be used for building safe and secure Cyber-Physical Systems. The main building blocks of CRYSTAL are introduced in [1] and [2]. The cornerstone of CRYSTAL is its architecture based on the MAPE-K[2] feedback loop [3] where we have components to monitor the system, analyze the behavior of the system, plan accordingly, and actuate necessary actions.

In CRYSTAL, we build an actor-based model of the system and the attack models, perform security analysis at design-time using formal verification in the form of model checking, and find the attacks that may jeopardize the system safety and security [1]. Then, abstract the state space generated by model checking to create an abstract behavioral model (Tiny Digital Twin) of the system, and use it at runtime [2]. Based on our experience, the actor model matches the domain of reactive and Cyber-Physical Systems well and hence the modeling becomes natural in these domains [4, 5, 6]. Using CRYSTAL, we are able to systematically model and check complex coordinated attacks. In the following, we elaborate the details of our framework, and explain the stages of the security process in various CPS applications.

CRYSTAL Contributions. CRYSTAL is designed to complement, not to replace, an industrial cybersecurity program. It provides the industrial system development with an opportunity to identify areas where existing processes can be strengthened. The highlight of CRYSTAL is addressing cybersecurity concerns by deploying a monitor which uses a reference model of the system for attack detection in the feedback control loops of CPSs. The style is in the form of MAPE-K architecture, where the reference model acts like model@runtime for analysis and planning. A formally verified actor model is used to build the reference model as a Tiny Digital Twin. By cross-referencing the actual behavior of the system against the Tiny Digital Twin, we can identify a good percentage of discrepancies and take appropriate corrective measures to ensure reliability and security of the system. We use a concept of logical time, and a technique for logical and physical time alignment to be able to monitor a CPS.

We use Timed Rebeca as an actor-based modeling language supported by a model checking tool [7, 8] to model the behavior of the cyber-physical systems. We use Lingua Franca (LF) [9] to build an executable model of

[1]CRYSTAL stands for CybeR-physical sYstem SecuriTy AnaLysis

[2]Monitor, Analyze, Plan, Execute - Knowledeg-base

the system. LF is a programming language based on the Reactor model of computation [10] for building CPSs. LF deploys a mechanism to synchronize the logical time (defined in the Timed Rebeca model) with the physical time in the system.

The CRYSTAL framework is developed incrementally. In [1], we start by developing a Timed Rebeca model from the system specification and verify the safety and security requirements using Afra model checker [11]. In [1], we show how to model the components of Cyber-Physical Systems as actors in Timed Rebeca, and define interactions between the components as messages passed between the actors. We build attack models based on STRIDE threat modeling [12] as the guideline for defining attack scenarios. We the show how to verify the safety and security requirements using Afra model checker [11]. We use the Secure Water Treatment System (SWaT) to show the modeling and formal verification. In [2], we propose a monitor that uses a Tiny Digital Twin to detect false sensor data and faulty control commands. The Tiny Digital Twin is in the form of a state transition model. The monitor checks whether the observed data and commands are consistent with the transitions in the Tiny Digital Twin. In [13], we provide a formal foundation for mapping the state space of a Timed Rebeca model generated by Afra to the input of mCRL2 *ltsconvert* tool [14], by which we abstract away non-observable actions from the state space while preserving trace equivalence. In [2] and [13] we use the Temperature Control System (TCS) case study to demonstrate the methods. In this paper, we provide a complete overview of CRYSTAL, elaborating more on the runtime monitoring phase, and highlighting the details of the systems under study, attacks that can be detected. Moreover, the Pneumatic Control System is presented in this paper as an example where the distributed controllers communicating through insecure channels.

The CRYSTAL framework comprises modeling and programming for constructing cyber-physical systems, security analysis through model checking, and runtime monitoring. We have designed three stages that can be used to enhance cyber-resilience in CPS applications.

- We use the actor-based language, Timed Rebeca, for modeling and a mapping technique to generate executable code from the model.
- We build attack models and define security and safety properties, and we use them in model checking.

- We create a Tiny Digital Twin; for that we map the state space generated by the model checker into a format that can be the input of mCRL2 ltsconvert tool, and then abstract it using the reduction techniques of the tool.

- We perform runtime monitoring to detect attacks by checking the consistency of the behavior of the operational system with the Tiny Digital Twin.

We demonstrate the details of the methodology to detect attacks using Pneumatic Control System (PCS), Temperature Control System (TCS), and Secure Water Treatment System (SWaT). The PCS and SWaT systems are distributed control systems whereas TCS is a centralized control system. Aligning logical and physical time, enables us to perform the monitoring at runtime. Relying only on the logical times defined in the model is not a realistic assumption at runtime. We model both periodic and trigger sensors, which are two different types of sensors used in PCS and TCS. The use of different sensor types in PCS and TCS systems highlights the importance of adapting the modeling approach to the specific characteristics of each system. We show how we can model the impact of the environment on the TCS system functionality by using nondeterministic assignment for state variables. We highlight the multiple incoming connections and how we use priorities for events in the development of the SWaT case study in LF. We demonstrate the detection capability of the monitor and discuss the detection rate for each case study while enumerating possible attack scenarios.

Outline. We describe CRYSTAL framework and methodology in Section 2. We introduce Timed Rebeca and Lingua Franca in Section 3. We present our approach for modeling system and defining attacks in Section 4. In Sections 5, 6 and 7, we follow CRYSTAL methodology on three case studies and show the results of security analysis at both the design-time and runtime. Section 8 covers the related works. The conclusion of this work and future directions are discussed in Section 9.

2. The Framework

We realize three stages in the CRYSTAL framework: modeling and code generation, design-time security analysis, and runtime monitoring, as shown in Figure 1.

Modeling and code generation. We build a Timed Rebeca model [15, 8] to represent the behavior of a CPS. The approach we use for creating the Timed Rebeca model depends on the system. We may be building a system from scratch, or we may be dealing with an already existing system. We may start from the specification documents of the system and UML diagrams, or if the company is using Microsoft STRIDE [12] then we may have the Date Flow Diagrams (DFD) of the system. In the model it is enough to capture the main functionalities and behavior of the system in order to produce the correct output based on the inputs. More discussion on this topic may be found in [5, 16, 6].

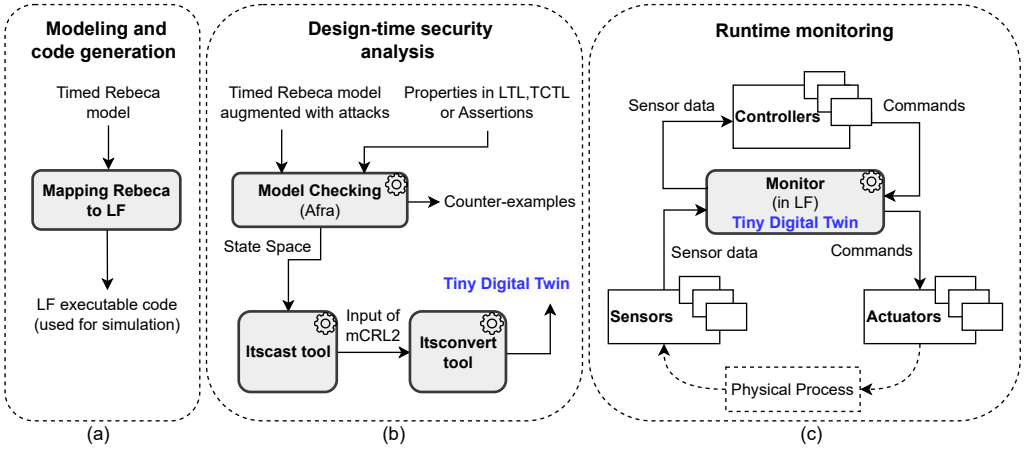

Figure 1: Three stages in the CRYSTAL framework. In stage (a), we build a Timed Rebeca model of the system. Then we map the Timed Rebeca model to an LF executable code. This LF code is used in simulation when checking the performance of the monitor at runtime (in stage (c)). In stage (b), we build a Timed Rebeca model that is augmented with attacks to find potential vulnerable points at design-time. We check the counter-examples generated by Afra to identify the trace of events leading to a failure. To build the Tiny Digital Twin, the state space is generated where none of the attacks are activated in the Timed Rebeca model. We use our ltscast tool to map the state space to the input format of ltsconvert tool of mCRL2. Then, the Tiny Digital Twin is built using ltsconvert tool. In stage (c), the monitor (written in LF) uses the Tiny Digital Twin of the system to detect cyberattacks at runtime.

We use LF to develop an executable code for a CPS. In LF, you may choose a target language like C or C++ for writing the body of reactors. Reactors are very close to Rebeca in syntax and semantics as shown in [17], and this enables us to effectively generate an executable target code from Timed Rebeca models. In the mapping between Timed Rebeca and LF, each

reactor in LF is mapped to a reactive class, and each reaction is mapped to a message server in Rebeca. In LF we build the bindings between inputs and outputs explicitly in the connection part of the program. For the timing issues, there is an after keyword in LF that has the same semantics as in Timed Rebeca.

Design-time security analysis. To check the security vulnerabilities at design-time we need attack models to be combined with the system model. The attack scenarios can be built based on the referenced guidelines in the security domain, e.g., STRIDE threat model. The attack models mimic real cyber and physical attacks and target the assets of the system to compromise their security properties or intended functionally, i.e., attacks on communication and components to achieve communication outage or reveal secret data.

We write the correctness properties from system security requirements, and feed both the combined model of the system and attacks, and the property file to the Rebeca model checking tool (Afra) [11] in order to evaluate the system tolerance against the attacks. We use the state space which is the output of Afra to build an abstract behavioral model of the system. The abstract behavioral model is used as the Tiny Digital Twin to help us in detecting the attacks in the runtime monitoring. We use the mCRL2 ltsconvert tool [14] to generate the Tiny Digital Twin while preserving the trace equivalency.

Runtime monitoring. During operational phase of the system, the Tiny Digital Twin is used within a monitor to detect cyberattacks on sensor data and control commands, and identify compromised components such as controllers. The monitor is strategically positioned between the control part and the sensor and actuator components in CPS applications as shown in Figure 1.(c). It observes the visible inputs and outputs of the controllers, traverses state transitions in the Tiny Digital Twin, and detects any misbehavior occurring during system operation. To protect the system against attacks and prevent damage, the monitor drops control commands that are not consistent with the state transitions in the Tiny Digital Twin. Using the Tiny Digital Twin, and the knowledge of the correct and secure functionality of the system, enables the monitor to validate the sequence of actions and the completion time of processes. The monitor is developed using LF language and has the same functionality in different CPS applications. It adjusts the input/output ports based on the number of sensors and actuators of the system.

3. Background: Timed Rebeca and Lingua Franca

In this section, we provide an overview on Timed Rebeca [7], and describe Lingua Franca programming language [18].

Timed Rebeca. Rebeca (Reactive Object Language) [19] is an actor-based language for modeling and formal verification of concurrent and distributed systems. An actor, called *rebec* (reactive Object), is an instance of a *reactive class*. Rebecs communicate via asynchronous message passing, which is non-blocking for both sender and receiver. Timed Rebeca, as an extension of Rebeca, has a notion of logical time. The logical time is local times of actors synchronized among all actors, that can be seen as a global time. Each actor has a set of variables that stores values, a set of methods (called *message servers*) and a message bag to store the received messages along with their arrival times and their deadlines. The actor takes a message with the least arrival time from its bag and executes the corresponding *message server*. The actor can change the values of its variables and send messages to its *known actors* while executing a message server. In Timed Rebeca, the primitives *delay* and *after* are used to model the progress of time while executing a message server.

Timed Rebeca is supported by Afra model checker tool [11]. Afra generates the state space of the Timed Rebeca model, in which states contain the local state of all actors and the logical time, and transitions represent three types of possible actions including taking a message from the message bag, executing the corresponding message server of the enabled actor, and progressing the logical time of the model. An approach based on a *shift-equivalence relation* is proposed in [8] to make the state space of a Timed Rebeca model bounded. Two states are in the shift-equivalence relation when all the elements of both states have the same value except for the elements related to time (like the current time value, and the time tags on the messages in the queues including deadlines). The elements related to time can be different but they should all have the same difference (shift) in their amount.

***Lingua Franca* (LF)**. Lingua Franca is a coordination language based on the Reactor model for programming CPSs [9, 20]. A Reactor model is a collection of *reactors* (like rebecs in Rebeca). A reactor has one or more routines that are called *reactions* (like message servers in Rebeca). Reactions define the functionality of the reactor, and have access to a *state* shared with other reactions, but only within the same reactor (similar to Rebeca). Reactors have named (and typed) *ports* that allow them to connect

to other reactors. Two reactors can communicate if an *output* port of a reactor is connected to an *input* port of the other reactor. The usage of *ports* establishes a clean separation between the functionality and composition of reactors; a reactor only references its own ports. Reactions are similar to the message handlers in the actor model. Reactions are triggered by discrete events and may also produce them (similar to handling a message and sending a message). An event relates a value to a *tag* that represents the logical time at which the value is present (similar to a time tag for a message). An event produced by one reactor is only observed by other reactors that are connected to the port on which the event is produced. Events arrive at input ports, and reactions produce events via output ports.

In LF, the logical time does not advance during a reaction. A reactor can have one or more *timers*. Timers are like ports that can trigger reactions. A timer has the form *timer name(offset, period)* that once triggers at the time shown by *offset* (if *offset* is zero, then the timer triggers at the start time of the execution), and then triggers periodically according to the *period*. LF has a built-in type for specifying time intervals. A time interval consists of an integer value accompanied by a time unit (e.g., *sec* for seconds or *msec* for milliseconds). Timers are used for specifying periodic tasks, which are very common in embedded computing and CPSs. Each LF code contains a *main reactor* that is an entry point for the execution of the code. The mapping of Timed Rebeca to Lingua Franca and reverse, including the timing features, is a natural mapping that is discussed in [17, 21].

4. Building the Rebeca Models and Attacks

In this section, we describe the method to model the behavior of the system and define attack scenarios in Timed Rebeca. We consider each component and physical process as an actor in CPS applications. We realize four categories of actors in the Rebeca model including controllers, sensors, actuators, and physical processes. Generally, the interaction scenarios between these actors follow a closed-loop feedback. Sensor observes the physical component's status, and sends the sensed data to the controller denoting the state of the physical component. Based on the received sensed data, the controller sends the control command to the actuator, and the actuator performs the actual physical change. In real cases, we may have different types of actors in each category (e.g., temperature sensors, speed sensors, etc.), and each type may be defined by a distinct reactive class.

Generally, the continuous behavior of physical components is expressed using differential equations like in Hybrid Automata [22]. We abstract the continuous behavior and only model the discrete jump transitions among the states (states are called control modes in hybrid automata). We model the progress of time in each state in Timed Rebeca. In each actor representing a physical component, we use state variables to model different states. In the remainder of this section, we will present the method to define attacks.

4.1. Attack Modeling

According to [1], the malicious behavior on communication channels and components are considered in three cases: (1) an attacker targets the communication channel between two components through injecting malicious messages, (2) an attacker manipulates the internal behavior of one or more components e.g. through malicious code injection, and (3) one or more attackers perform a coordinated attack to launch malicious behavior on both the communication channels and the components. To illustrate these cases, we define three attack schemes.

Scheme-A: Attack on Communication Channels can happen when an attacker sends malicious messages through the channels between the controller and the sensors/actuators. These messages may mislead the receiver and cause a system failure. For example, as depicted in Figure 2 for Scheme-A, an attacker can compromise the channel between the sensor and controller and send false sensor data that makes the controller give the wrong command to the actuator, causing unexpected results. In the Timed Rebeca model, a reactive class is defined to model the attacker's behavior. This class sends malicious messages at an appropriate time to damage the system. To perform exhaustive security check, a set of Timed Rebeca models can be developed that contains one or more attackers that target different channels at different injection times during the system operation. These Timed Rebeca models are inputs of the security analysis using model checking.

Scheme-B: Attack on Components indicates a situation in which a number of components are compromised and do not function correctly. Attackers may have direct access to the components and perform physical attacks on them. They may damage the code in some sensors/actuators or inject malicious code into the controllers. For example, as Figure 2 shows for Scheme-B, an attacker may compromise controllers and perform actions over a physical process different from the desired plan. This scheme is modeled in Timed Rebeca as modified message server inside the reactive class corresponding to the

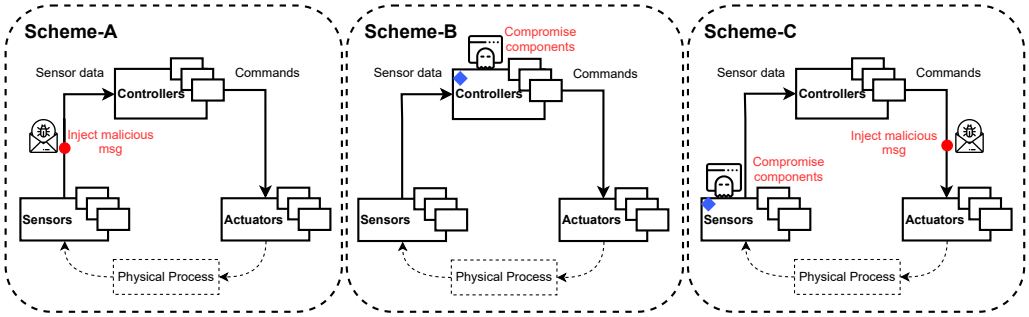

Figure 2: Scheme-A, Scheme-B, and Scheme-C in Timed Rebeca model for security analysis of CPS applications (adapted from [1]). The red circles show attacks on communication channels and the blue diamonds indicate attacks on components. For example, in Scheme-A the attacker injects malicious data into the communication channel between a sensor and the controller. In Scheme-B, the attacker compromises controllers, and in Scheme-C, there is an attack that is performed in a coordinated way.

target component. This message server models the incorrect functionality.
Scheme-C: Combined Attack is a combination of the previous two attack schemes. In this scheme, attackers can compromise both the system components and communication channels by coordinating their attacks. Scheme-C in Figure 2 illustrates a system with the presence of an attack in which an attacker injects faulty command into the channel between the controller and the actuator, and compromises the sensor to report false sensor data to the controller. The modeling of this scheme would include various combinations of the defined attackers and compromised components as actors in a Rebeca model. We can choose many kinds of attack scenarios with the assumption of compromised network or components in Rebeca model and check the attacks' damage on the CPS applications.

4.2. Attack Classification

STRIDE[1] categorizes threats and corresponding security objectives for systems. Table 1 classifies significant CPS attacks (reported in [23, 24, 25]) based on the STRIDE categories. Attacks exploit communication and component vulnerabilities in *Scheme-A* and *Scheme-B*. We are able to model these attack scenarios using our methodology.

[1]The acronym STRIDE stands for **S**poofing, **T**ampering, **R**eputation, **I**nformation Disclosure, **D**enial of Service, and **E**levation of Privilege.

Table 1: Attack Classification using STRIDE model [1].

Threat Type (Security Objective)	*Cyber and Physical Attack*	*Scheme-A*	*Scheme-B*
Spoofing (Authentication)	Masquerade attack		[24]
	Packet spoofing attack	[25]	
Tampering (Integrity)	Man-in-the-middle (MITM)	[24]	
	Injection attack	[25]	[25]
	Replay attack	[24]	
	Malware (Virus or Worms)		[25]
	Physical attack	[25]	[23]
Reputation (Non-Repudiation)	On-Off attack		[23]
Information Disclosure (Confidentiality)	Eavesdropping	[24]	
	Malware (Spyware)		[25]
	Side-channel attack		[25]
	Physical attack	[25]	[23]
Denial of Service (Availability)	Resource exhaustion attack	[24]	[25]
	Interruption attack	[24]	
	Malware (Ransomware)		[25]
	Physical attack	[25]	[23]
Elevation of Privilege (Authorization)	Malware (Rootkit)		[25]

The attacks that compromise the communication channels belong to the following category. In *Spoofing attack*, the attacker transmits a message with a spoofed identity into the network. *Man-in-the-middle (MITM)* requires the attacker to put herself in between two communicating parties and change the messages. To launch MITM attack, the attacker impersonates herself as one of the targeted parties. *Injection attack* indicates that the attacker injects invalid messages into the network (i.e., packet injection). *Replay attack* is an intentional repetition of sending a message to mislead the receiver. *Eavesdropping attack* takes advantage of unsecured channels to steal the information transmitted over the network. *Side-channel attack* is an attack in which the attacker uses her own technical knowledge of the system to compromise the system security. *Resource exhaustion attack* represents a situation that the network resources are overwhelmed by a flood of messages transmitted from the attacker. *Interruption attack* makes a service unavailable for legitimate use, and *Physical attack* aims to damage a communication link.

The attacks that compromise the components belong to the following category. *Masquerade attack* refers to a situation where the attacker impersonates herself as one of the communicating parties. *Injection attack* is used

by an attacker to inject a malicious code into a component (i.e., code injection). *Malware* is a malicious software designed to manipulate the behavior of components. *Side-channel attack* is an attack in which the attacker gains knowledge about the system by observing the behavior of some component(s). Finally, *Physical attack* manipulates some component(s) physically.

5. Pneumatic Control System (PCS)

Pneumatic Control System (PCS) is a control system that regulates the movement of mechanical components, such as cylinders, robotic arms or conveyor belts, in multiple directions. The system is widely used in various safety-critical industrial applications, including manufacturing and automotive industries due to its high reliability and low maintenance requirements. We use a PCS described in [26] to explain CRYSTAL.

The PCS presented here has two cylinders, CylinderA and CylinderB, as shown in Figure 3. Each cylinder is controlled by a dedicated controller to regulate the movement in either left-right or up-down directions. The timing of the movement for cylinders can differ based on the direction of the movement. The controllers are responsible for coordinating the movements in the correct sequence and timing that involve pick-and-place operations. The motion plan is moving the cylinders from the initial position (location X) to the target position (location Y), and then moving back to the initial position. In this case, each movement takes 2 units of time. The desired sequence of movements of the cylinders is as follows: (1) CylinderB moves down, (2) CylinderB moves up, (3) CylinderA moves right, (4) CylinderB moves down and (5) then up, (6) CylinderA moves left.

5.1. PCS Timed Rebeca model

The Timed Rebeca model for the PCS case study is depicted in Listing 1. The model also contains parts for modeling the compromised version of sensors, and an attacker (explained in Section 5.2). In this model, there are six reactive classes ControllerA, SensorA, CylinderA, ControllerB, SensorB, and CylinderB. Each controller receives the information about the position of its own cylinder from the corresponding Sensor, using the message server getsense (see line 10). The controllers also receive the information about the other cylinder movements using the message server getctl (line 20). The controllers decide whether to move the cylinder based on the current status

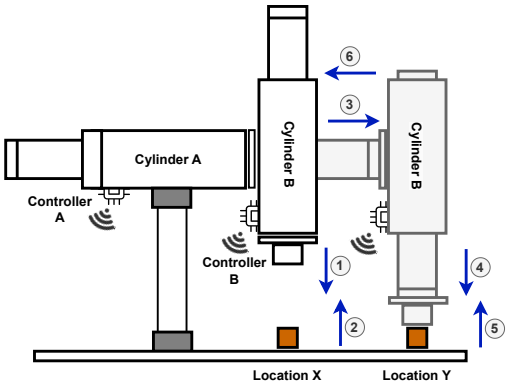

Figure 3: PCS with two cylinders (adapted from [26]). The cylinders pick up a particle from location X and place it in location Y.

and desired movement (lines 11 to 19). The controllers send the motion commands 1 or -1 to regulate the movements (lines 15 and 19). The sensors in this system are the trigger sensors and simply serve as intermediaries between the cylinders and controllers, reporting location information (lines 22 to 27) when cylinders touch initial location or target location. Sensing the location of cylinders is modeled using a message server status in both cylinders that updates the motion and reports it to the respective Sensor, which then forwards the location to the corresponding Controller (line 34). The message server actuate receives a motion command from the Controller to handle the movements (line 35). The time of the movement for each cylinder is modeled using after primitive (line 36). In this model, the duration of each movement is 2 units of time. We assume that ControllerA starts its linear motion from the left at the top of location X. The initial and end locations (value 0 for left-top, 2 for right-top and -2 for right-down) for both cylinders are set through environmental variables (see line 1).

5.2. PCS attack modeling in Timed Rebeca

The Timed Rebeca model of the system is augmented with different types of attacks that can be launched to test the resilience of the system. The attack scenarios that are modeled in this case study include *compromised sensors*, *compromised cylinders*, and *injection attacks* on the communication channels between controllers as shown in Figure 4. In addition, the *combined attacks* are performed by involving injection attack with the compromised version of sensors and cylinders to perform complex coordinated attacks.

```
env int cylinderAEndloc = 2; env int cylinderBEndloc = -2; //environment variables
env boolean sAComp = false; env int sAComp_time = 6; env int sAmalMsg = 0;
env boolean inj_attk = false; env int attTime = 4;
env int malMsg = 0; env int chl = 1;
...
reactiveclass ControllerA(5){
    knownrebecs{ CylinderA cylA; ControllerB cntlB;}
    statevars{boolean locBisUP, locAisLeft;}
    ControllerA(){ locBisUP = false; locAisLeft = true;}
    msgsrv getsense(int locA) { //locBisUP:true means that CylinderB is up
        if(locBisUP) {
            if(locAisLeft) {
                if(locA == cylinderAEndloc) {
                    cntlB.getctl(locA); locAisLeft = false; locBisUP = false;
                } else { cylA.actuate(1);}
            } else if (!locAisLeft) {
                if(locA == 0) {
                    cntlB.getctl(locA); locAisLeft = true; locBisUP = false;
                } else { cylA.actuate(-1); } } } }
    msgsrv getctl(int locB){ if (locB == 0) { locBisUP = true; } }
}
reactiveclass SensorA(5){
        knownrebecs{ ControllerA ControllerA;}
        SensorA(boolean compromised, int compTime, int msg){
                if (compromised) { self.getloc(msg) after(compTime);} }
        msgsrv getloc(int loc) { ControllerA.getsense(loc);}
}
reactiveclass CylinderA(5){
        knownrebecs{SensorA SensorA;}
        statevars{ int loc, motion;}
        CylinderA(boolean compromised, int compTime, int msg){
                loc = 0; motion = 0; self.status();
                if (compromised) { self.actuate(msg) after(compTime); }}
        msgsrv status() { loc = loc + motion; // left to right on x-axis
            if(loc == 0 || loc == cylinderAEndloc){
                SensorA.getloc(loc);  motion = 0; } self.status() after(2); }
        msgsrv actuate(int rate) { motion = rate;}
}
reactiveclass ControllerB(5){...}
reactiveclass SensorB(5){...}
reactiveclass CylinderB(5){...}
reactiveclass Attacker(3){//injects false messages in channels between controllers
        knownrebecs{ ControllerA cntlA; ControllerB cntlB;}
        Attacker(boolean inj, int channel, int msg, int attktime) {
         if(inj){  if (channel == 1) { self.chlBA(msg, attktime);}
                   if (channel == 2) { self.chlAB(msg, attktime); } } }
        msgsrv chlBA(int msg, int attktime){ cntlA.getctl(msg) after(attktime); }
        msgsrv chlAB(int msg, int attktime){ cntlB.getctl(msg) after(attktime); }
}
main{
    CylinderA cylA(SensorA):(cAComp,cAComp_time,cAmalMsg);
    ControllerA ControllerA(cylA, ControllerB):();
    SensorA SensorA(ControllerA):(sAComp,sAComp_time,sAmalMsg);
    ... // ControllerB and SensorB instances
    Attacker attacker(ControllerA, ControllerB):(inj_attk, chl, malMsg, attTime);}
```

Listing 1: A part of the Timed Rebeca model augmented with attacks for the PCS case study.

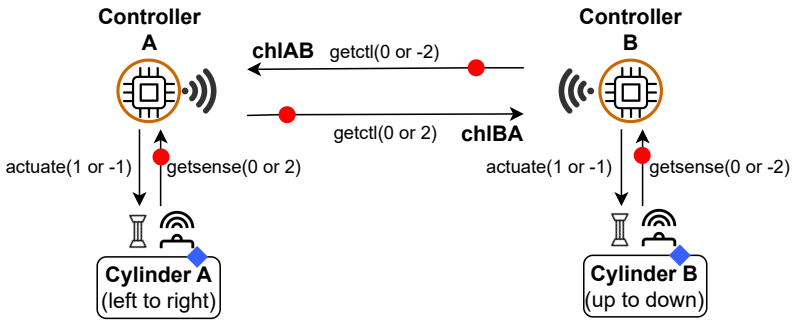

Figure 4: The sensor data and control commands in the PCS case study. The messages are transmitted between controllers in order to regulate the movements. The possible attack points for performing attack scenarios are depicted with red circles and blue diamonds to show attacks on communications or components, respectively.

In the following, we explain the attacks augmented in the Timed Rebeca model. The *compromised sensors* are where SensorA and SensorB can be compromised by an attacker. In Timed Rebeca model, this attack scenario is modeled by setting a flag in the input parameter of the constructor that changes the mode of the sensor component to compromised (see line 24 in listing 1). When the sensor is compromised, it sends a malicious message to its corresponding controller at a specified time. The SensorA(compromised, compTime, msg) shows how it is activated in the constructor (line 24). The parameter msg indicates the current location of the cylinder being sensed, which may be different from the actual location. This can mislead the controller to take the wrong decision for the control motion and cause a failure in the system. The *compromised cylinders* are where CylinderA and CylinderB can be compromised by an attacker. Similar to the attack on sensors, they are modeled by setting a flag in the input parameter of the constructor of the reactive classes (line 31). For example, when CylinderA is compromised, it receives a malicious command from an attacker at a specified time (i.e., CylinderA(compromised, compTime, msg))(line 45). The msg can be either to move left or right, causing the cylinder to move to a different location than intended and compromise the system. The *injection attacks* show that the system is also susceptible to injection attacks, where an attacker can inject a malicious message into the communication channels, i.e., chlAB and chlBA, between two controllers. This can cause the controllers to take the wrong decision for the movements and send wrong

motion commands. We define the actor Attacker to perform injection attacks on the system (line 42). The channels chIAB and chIBA are compromised and a malicious sensor message is injected at a specified time (i.e., Attacker(inj, channel, msg, attktime))(lines 45 to 49). In order to test for possible complex attack scenarios, we must generate combinations of different values for both the input parameters of the Attacker and the Compromised components, and verify the model for each combination. To automate this process, we develop a Python script for generating input values and collecting verification results. This approach is similar in nature to the automated verification technique that uses symbolic modeling and constraint solving. The complete model of the system and the written python codes are available on GitHub [27].

5.3. PCS safety properties

We define safety properties to catch unsafe and undesirable movements of the cylinders. To specify the properties, we use *assertions*. The Timed Rebeca model for the PCS satisfies all the properties in Listing 2 when none of the attacks are activated. If the model checker detects that a safety property is not satisfied, it provides the modeler with a counter-example that outlines the sequence of events leading to the violation. This sequence of events can be used to determine the steps of the successful attack scenario when the compromised components or injection attacks are activated.

The properties shown in Listing 2 are defined using the values of the variables loc and motion for two cylinders in the Timed Rebeca model. The variable loc keeps the location information of the cylinder at each state and the variable motion indicates the motion command issued by the controller. The property safety_prop1 is written to ensure that CylinderA cannot move to the right (motionR) or left (motionL) while CylinderB is located at the bottom (locBb). The properties safety_prop2 and safety_prop3 ensure that both cylinders do not move diagonally. In these safety properties, CylinderB cannot move up (motionU) or down (motionD) while CylinderA is moving to the right (motionR) or left (motionL). The properties safety_prop4 and safety_prop5 ensure that CylinderA and CylinderB, respectively, only have motion between the initial position and the end position in location X or Y.

```
property {
    define {
        locXa = cylA.loc == 0; locYa = cylB.loc == 0;
        locXb = cylA.loc == 2; locYb = cylB.loc == -2;
        motionR = cylA.motion == 1; motionL = cylA.motion == -1;
        motionU = cylB.motion == 1; motionD = cylB.motion == -1;
        locXbstuck = cylA.loc == 3; locXastuck = cylA.loc == -1;
        locYbstuck = cylB.loc == -3; locYastuck = cylB.loc == 1;
    }
    Assertion {
        safety_prop1: !((motionR && locYb) || (motionL && locYb));
        safety_prop2: !((motionR && motionU) || (motionL && motionD));
        safety_prop3: !((motionL && motionU) || (motionR && motionD));
        safety_prop4: !(locXastuck ||locXbstuck);
        safety_prop5: !(locYastuck ||locYbstuck);
    }

}
```

Listing 2: The safety properties for the PCS case study.

5.4. PCS security analysis

Table 2 and Table 3 show the results of the analysis based on the model checker outputs. In our experiments, we consider the number of false sensor data and faulty control commands as the number of *compromising* and *injection* attacks during a predefined system execution period (28 seconds). Among the total number of 2977 attacks on sensor data and control commands (i.e., via Compromised version of the components), 355 attack scenarios successfully violated the safety properties. Similarly, out of the 60 injection attacks, 28 attack scenarios successfully violated the safety properties safety_prop1, safety_prop2, and safety_prop3. Therefore, the total number of attacks that violated the properties is 383 (i.e., total_successful_attacks) out of 3,037 attacks.

Table 2 uses the following notation: sACompromised indicates that SensorA is compromised, while cACompromised indicates that CylinderA is compromised, and so on for the other components. The high-risk component is SensorB since preforming attack scenarios using sBCompromised shows the highest total number of failures for the properties (i.e. 140, highlighted in pink). This means that SensorB is a more vulnerable point and needs to be protected against potential attacks. Additionally, safety_prop3 is violated more than other properties during attacks, indicating that this function in the system behavior is more sensitive (highlighted in blue). We also observe that some properties have the higher number of violations per attack (highlighted in gray).

Table 2: The failure of properties for the compromising and injection attacks

	safety_prop1	safety_prop2	safety_prop3	safety_prop4	safety_prop5	total_successful_attacks
cACompromised	5	14	16	3	0	38
cBCompromised	0	19	17	0	13	49
sACompromised	2	44	78	0	4	128
sBCompromised	20	50	31	0	39	140
injection	6	10	12	0	0	28
total_fails_prop	33	137	154	3	56	383

Table 3 shows the results for the combined attack scenarios. We consider the compromising and injection attacks that are described in the attack scenarios above, and combine those scenarios pairwise where they do not violate the safety properties without combination. The outcomes show the pairs of attacks that result in successful coordinated attack scenarios. Out of the 8010 combined attack scenarios, 281 cases successfully violate the safety properties. We show the property with the higher number of violations (highlighted in gray) for each pair of attacks. The most effective attack is the combination of the attack on SensorB and the injection of false data into the channels (highlighted in pink).

Table 3: The failure of properties for the combined attacks

	safety_prop1	safety_prop2	safety_prop3	safety_prop4	safety_prop5	total_successful_attacks
sACompromised+injection	0	6	36	5	0	47
sACompromised+sBCompromised	2	24	15	0	4	45
sBCompromised+injection	8	33	10	0	24	75
cBCompromised+injection	0	13	10	0	2	25
cBCompromised+sACompromised	0	9	17	0	0	26
cACompromised+sBCompromised	0	0	5	0	0	5
cACompromised+injection	0	4	2	3	0	9
cBCompromised+sBCompromised	5	14	8	0	11	38
cACompromised+sACompromised	0	0	2	0	0	2
cACompromised+cBCompromised	1	0	2	0	0	3
total_fails_prop	16	103	107	8	41	281

An example of a combined attack scenario is sBCompromised + injection where the attack violates the property safety_prop2. As shown within Rebeca IDE in Figure 5, the attack scenario in the Timed Rebeca model involves an attacker where it injects the false sensor data 0 into the channel between the controllers at attktime = 6 and sBCompromised transmits the false sensor data 0 to ControllerB at sBComp_time = 12. In this combined attack scenario, as shown in Figure 6, (1) the attacker waits until 6 units of time and then injects the false sensor data getctl(0) into the channel towards ControllerB. The false sensor data 0 indicates the movement of CylinderA is completed and ControllerB can actuate CylinderB. (2) ControllerB sends the motion command actuate(1) to actuate CylinderB. (3) ControllerB gets the status of CylinderB from the compromised sensor where it reports that CylinderB

is moved down. (4) ControllerB reports ControllerA that the movement of CylinderB is completed since it gets the false sensor data getsense(0). (5) ControllerA gets the sensor data getsense(0) from SensorA where it indicates CylinderA is on the left and also gets the report getctl(0) from ControllerB showing that the movement of CylinderB is completed. (6) ControllerA sends the motion command actuate(1) to actuate CylinderA to the right at time 12. The property safety_prop2 is violated when actuate(1) is sent from ControllerA as shown in the counter-example at the top right of the IDE in Figure 5.

The property safety_prop2 ensures that both cylinders must not move diagonally. Therefore, the combined attack scenario violates the safety property safety_prop2 at time 12 and the model checker generates the counter-example that indicates the events leading to the violated state. In this case, the injection attack and the compromised sensor attack are not successful attacks separately. In our approach, we can extend the attack combinations and evaluate the system security with various complex attack scenarios.

5.5. PCS Tiny Digital Twin and Monitoring

We employ a monitor to find inconsistencies at runtime. The monitor observes sensor data and control commands transmitted in the network and detects attacks using Tiny Digital Twin. Tiny Digital Twin is an abstract version of the state space generated by the model checker and is used by the monitor to catch inconsistencies. The monitor walks over the model to check whether the sensor data and control commands are consistent with the state transitions in the Tiny Digital Twin.

The actions on the transitions in the state space that are not visible to the monitor (and the controller), are known as non-observable actions. We developed a tool, *ltscast*, to map the state space created by Afra into the input format of the mCRL2 *ltsconvert* tool [14]. Using the mCRL2 *ltsconvert* tool, we create the Tiny Digital Twin by abstracting away non-observable actions while preserving trace equivalence [13].

We create the Tiny Digital Twin for the PCS system by providing the *ltsconvert* tool with a list of labels that denote the the silent transitions (non-observable actions). In this system, the actions getsense, actuate and getctl are observable in the system behavior from the controller point of view, while actions status and getloc are non-observable. The resulting abstract model has 87 states and 120 transitions, while the original state space has 276 states and 439 transitions.

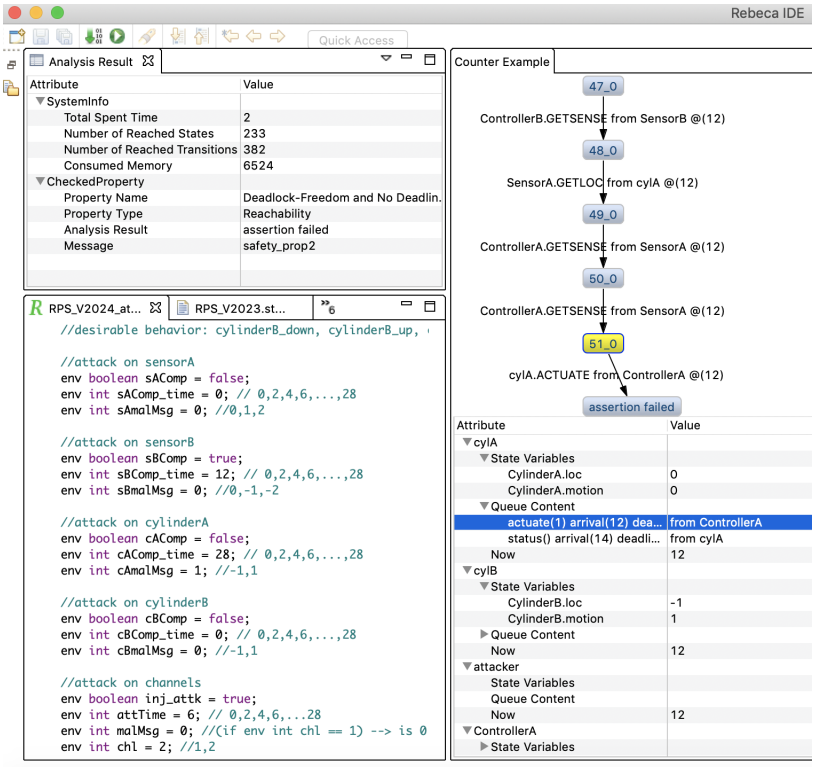

Figure 5: The counter-example generated by Afra model checker for the combined attack scenario cBCompromised + injection where attktime = 12. The property safety_prop2 is violated when actuate(1) is sent from ControllerA.

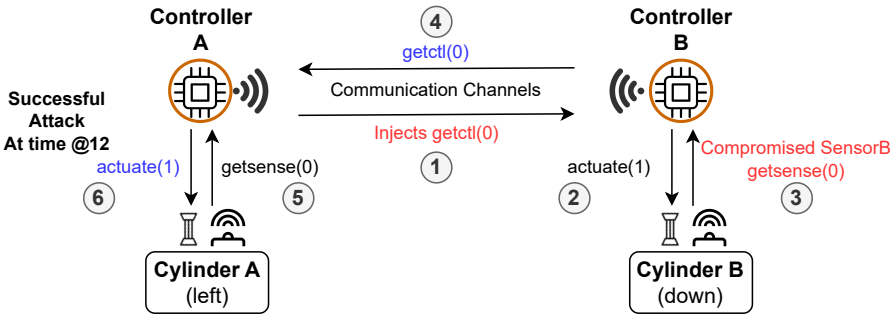

Figure 6: The sequence of events for the combined attack cBCompromised + injection where the attack succeeds at time attktime = 12 on the control system. The attack causes the system to subsequently generate getctl(0) and actuate(1), and violate the safety property safety_prop2.

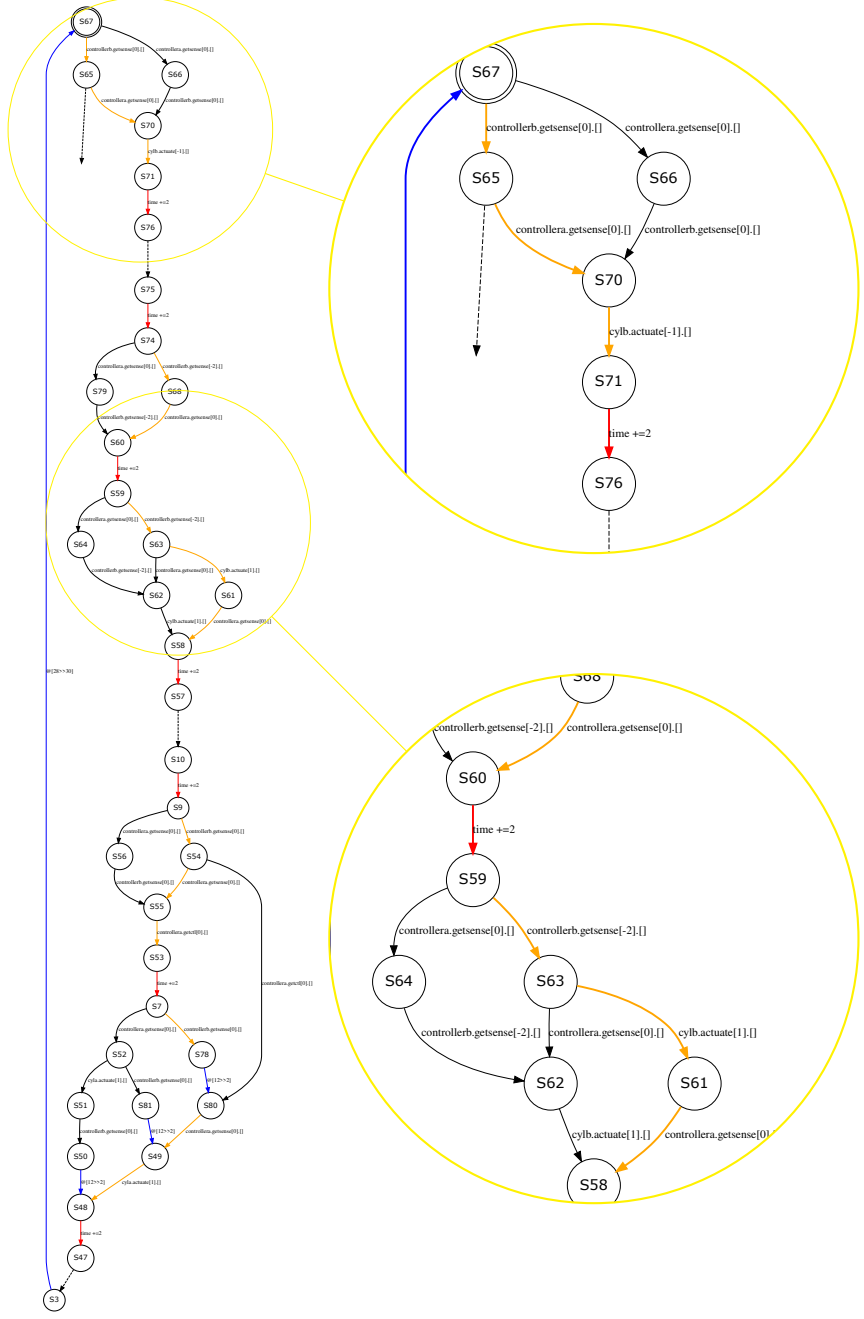

Figure 7: A subset of the state transitions in the Tiny Digital Twin of the PCS case study. A transition path that presents the actions to handle the desired movement of CylinderB is shown in the diagram (orange arrows).

Figure 7 shows a subset of state transitions of the Tiny Digital Twin. A transition path that presents the actions to handle the desired movement of CylinderB is shown in the diagram (orange arrows). SensorA and SensorB send getsense(0) to report that CylinderA is on the left and CylinderB is moved up (outgoing transitions from state S67 and state S65). Regarding the desired behavior of the system, ContollerB sends the motion command actuate(−1) to move CylinderB down (S70). Each motion in the movement of the cylinder takes 2 units of time (S71). SensorB updates the status of CylinderB to the controller (S76 to S75). The CylinderB moves in the linear motion and reachs the end location (i.e., getsense(−2)). ContollerB sends the motion command actuate(1) to move CylinderB back to the up (S74 to S61). ContollerB starts to send actuate(1) to CylinderA when it receives the report getsense(0) from SensorA and the message getctl(0) from ControllerB through the channel between the controllers (S61 to S48). The time shifting transitions between states are shown with blue arrows where they indicate the transition due to the shift-equivalence relation.

5.6. PCS Lingua Franca

In Timed Rebeca model and Tiny Digital Twin, we use logical time. However, the monitor deals with physical time in the real applications based on physical clocks. To synchronize logical time and physical time, we develop the monitor using Lingua Franca (LF). LF aligns these two timelines at runtime using a scheduler that monitors the local clock of each actor and delays processing the message until its measurement of physical time exceeds a threshold [20].

We use the mapping between Timed Rebeca and Lingua Franca presented in [17] and write a Lingua Franca code for the PCS case study. The code is presented in Listing 3 and its diagram is shown in Figure 8. The diagram shows reactors including their reactions for the system. To simulate the attacks, we modify the reactions in the reactors. This way, the reactors behave as Compromised components and respectively send false sensor data and faulty control commands on the output ports. In addition, the reactor Attacker is defined to inject false messages into the channel between the controllers.

Similar to the Timed Rebeca model of the system, the code implements all components of the system. The message servers getsense, getctl, status and actuate (see Listing 1) are mapped to the reactions of the corresponding reactors in LF (see Listing 3). The list of known rebecs in knownrebecs

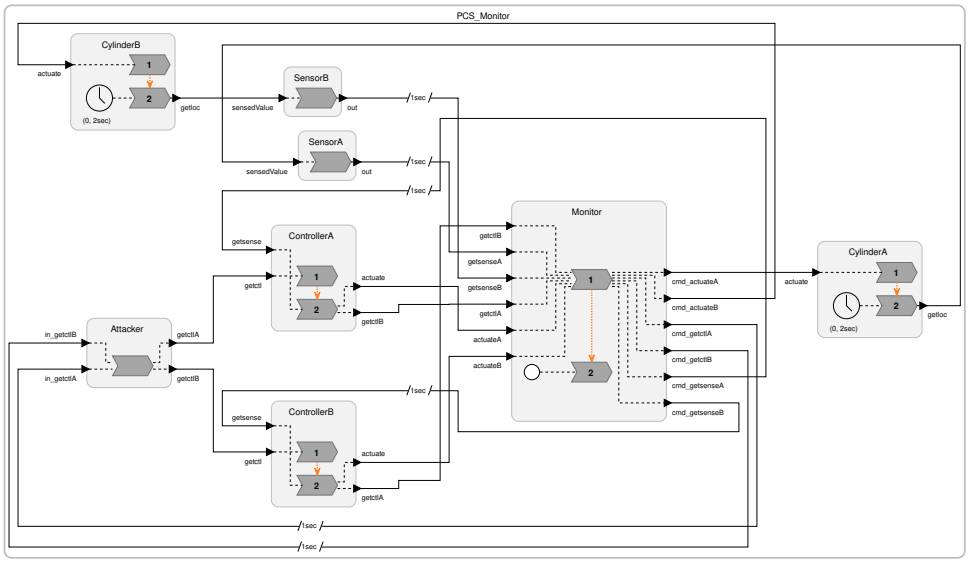

Figure 8: The diagram (built in Eclipse IDE) of the reactors in LF code for PCS case study including monitor and attacker reactors.

shows the number of output ports that are defined in the respective reactors. For example, the output ports actuate and getctlB in the reactor ControllerA are defined based on two known rebecs cylA and cntlB. The state variables locBisUP, locAisLeft, loc, and motion in the reactive classes ControllerA and CylinderA are mapped to the states in LF (see e.g., line 4). In Timed Rebeca, a message server of other reactive classes (or self) is called, and that is how the binding and the flow is realized. In LF, in the connection part of the main reactor, all the bindings are set by defining which input of which reactor is connected to which output of which reactor (see line 46 to 53).

As shown in Listing 3, the input ports getsense and getctl in the reactor ControllerA are defined to get sensor data and the report from ControllerB. Two output ports actuate and getctlB are defined to send values as the notion commands to CylinderA and reports the status of the cylinder to ControllerB (line 4). We can set the value of motion defined in the reactor CylinderA (line 29) by transmitting a value in the output port actuate. We set the value 1 to move the cylinder to the right (line 10), and -1 to move the cylinder to the left (line 14), and use the value of 0 to stop the motion of the cylinder.

The timing of the movement for the cylinders is defined using timer. The reactor CylinderA includes timer that is used to trigger the reaction status in

the cylinder reactor after every 2 seconds (i.e., timer status(0, 2 sec)). The reaction status changes the location loc of the cylinder based on the value of motion (line 28 to 41). The reactors ControllerB, SensorB and CylinderB are defined similarly for other components of the system (line 42 to 44).

We set the compromised version of SensorA and CylinderA using state variables (line 19). If the state variable compromised becomes true, the reactor SensorA behaves as the compromised version and compares the logical time provided in get_elapsed_logical_time() with the time value compTime. It sends msg into the output port when two logical times are the same (line 20 to 23). We have the similar implementation for the compromised version of CylinderA, CylinderB and SensorB (line 30 to 34).

The main reactor instantiates the components and binds their input and output ports to connect the components together. For example, we connect the output port out in the reactor SensorA to the input port getsenseA of the reactor Monitor. This way, the sensor data is transferred from the sensor to the Monitor (line 50). In the main reactor, the use of after indicates that a value reaches the input port getsenseA of the reactor Monitor after 1 unit of time.

5.7. Detection capability of the monitor in PCS

We consider those Compromised components and Injection attacks that successfully violate the safety properties at design-time (see Section 5.4, Table 2 and Table 3 for the total successful attacks) in evaluating the detection capability of the monitor at runtime. In our experiments, in developing LF code, we simulate 355 false sensor data and faulty actuation as listed in Table 2. We also simulate 28 injection attacks by defining Attacker and 281 combined attacks where the injection attacks are combined with sensor data and faulty commands.

We implement the monitor as a reactor in LF (i.e., Monitor.lf) (line 2). The reactor Monitor is imported to the LF code of the PCS case study. As shown in Figure 8, the reactor Monitor contains two reactions, one for loading Tiny Digital Twin and another for comparing input data with the transitions in the model. The code is compiled by the Lingua Franca compiler, and an executable file is returned. The monitor observes the sensor data and the control commands during the code execution and decides to drop or pass the commands to the cylinders. The complete LF code of the monitor and the components of the system are available in GitHub [27].

```
target Cpp {fast: false, threads: 1};
import Monitor.lf; //loads Tiny Digital Twin and compares inputs with labels on transtions
reactor ControllerA {
    input getsense:int; input getctl:int; output actuate:int; output getctlB:int;
    state locBisUP:bool(false); state locAisLeft:bool(true);
    reaction(getsense) -> actuate, getctlB {=
    if (locBisUP) {
        if (locAisLeft) {
            if(*getctl.get() == 2){ getctlB.set(*getctl.get());
                locAisLeft = false; locBisUP = false; } else { actuate.set(1); }
        } else if (!locAisLeft) {
            if(*getctl.get() == 0){
                getctlB.set(*getctl.get()); locAisLeft = true; locBisUP = false;
            } else { actuate.set(-1); } } } =}
    reaction(getctl) {= if (*getctl.get() == 0) { locBisUP = true; } =}
}
reactor SensorA {
    output out:int; input sensedValue:int;
    state compromised:bool(false); state compTime:int(0); state msg:int(0);
    reaction(sensedValue) -> out {=
    auto elapsed_time = get_elapsed_logical_time();
    auto elapsed_secs = std::chrono::duration_cast<std::chrono::seconds>(elapsed_time);
        if(compromised && elapsed_secs == std::chrono::seconds(compTime)){
            out.set(msg);
        } else { out.set(sensedValue.get()); }
    =}
}
reactor CylinderA {
    input actuate:int; output getloc:int; state loc:int(0); state motion:int(0);
    state compromised:bool(false); state compTime:int(0); state msg:int(0);
    reaction(actuate) {=
    auto elapsed_time = get_elapsed_logical_time();
    auto elapsed_secs = std::chrono::duration_cast<std::chrono::seconds>(elapsed_time);
        if(compromised && elapsed_secs == std::chrono::seconds(compTime)){ motion = msg;
        } else { motion = *actuate.get(); }
    =}
    timer status(0, 2 sec);
    reaction(status) -> getloc {=
        loc = loc + motion;
        if(loc == 0 || loc == 2){ getloc.set(loc); motion = 0; } =}
}
reactor ControllerB { ... }
reactor SensorB { ... }
reactor CylinderB { ... }
reactor Attacker { ... } //injections
main reactor PCS {
    ...
    ControllerA = new ControllerA(); ControllerB = new ControllerB();
    cylA.getloc -> SensorA.sensedValue; cylB.getloc -> SensorB.sensedValue;
    SensorA.out -> Monitor.getsenseA after 1 sec;
    ControllerA.actuate -> Monitor.actuateA; ControllerB.actuate -> Monitor.actuateB;
    Monitor.cmd_actuateA -> cylA.actuate; Monitor.cmd_actuateB -> cylB.actuate;
    Attacker.getctlA -> ControllerA.getctl; Attacker.getctlB -> ControllerB.getctl;
}
```

Listing 3: LF code for the PCS case study.

An example of runtime attack detection using Monitor is described below. Consider Tiny Digital Twin in Figure 7, and assume that the system is in state S74 and Compromised SensorA sends getsense(1) to the controller indicating CylinderA is moved to the right whereas the actual location that is sensed by SensorA is left. Upon receiving this false sensor data, Monitor produces an alarm and terminates the monitoring process because the data sent by the sensor is not the same with the labels on transitions from S74 to S60. As another example, assume the system is in state S70 and Compromised CylinderB moves up. Monitor does not detect this attack immediately because the transmitted command from the controller matches the one in Tiny Digital Twin while CylinderB moves up. Monitor detects the attack when it receives false sensor data getsense(-2) at S74. Monitor can drop the commands transmitted from the controller that are not consistent with the outgoing transition in Tiny Digital Twin. A coordinated attack can occur when Attacker injects getctl(0) while Compromised SensorB sends false sensor data getsense(0) to the controller at S54. In this coordinated attack, injecting getctl(0) into the channel is not successful since Monitor compares the order and timing of the events with the transitions in Tiny Digital Twin. Monitor compares getctl(0) with the outgoing transition at S54 and detects the false sensor data by comparing it with the data on the transition between S80 and S49.

Table 4: The detection capability of the monitor in the PCS case study

attack types	total_successful_attacks	detection capability
cACompromised	38	0
cBCompromised	49	0
sACompromised	128	128
sBCompromised	140	140
injection	28	28
sACompromised+injection	47	47
sACompromised+sBCompromised	45	45
sBCompromised+injection	75	75
cBCompromised+injection	25	8
cBCompromised+sACompromised	26	9
cACompromised+sBCompromised	5	2
cACompromised+injection	9	4
cBCompromised+sBCompromised	38	18
cACompromised+sACompromised	2	1
cACompromised+cBCompromised	3	0
total_detection_rate	664	505 (%76)

As shown in Table 4, the detection rate of the monitor can be calculated with respect to the detected/undetected attacks by the monitor. In this case study, out of the 664 attack scenarios (combined and single successful attacks), 159 attacks were not directly detected by the monitor, therefore the

detection rate is around 76 percent. The undetected attacks are related to the compromise of components cACompromised and cBCompromised, as well as their combination with other attacks, where the attack impact is reported by sensors after the attack succeed. The monitor can not immediately detect the attacks that directly affect the actuators and physical process, because the monitoring system relies on sensor data and control commands transmitted over the network to detect potential attacks, and compares them with the expected state transitions in the Tiny Digital Twin. To improve the ability to detect these types of attacks, it may be necessary to implement additional security measures, such as network segmentation, or endpoint protection solutions. These measures can help to identify and isolate compromised components or devices, as well as detect anomalous behavior that may indicate an ongoing attack.

6. Temperature Control System (TCS)

The temperature control system is responsible for maintaining the temperature of a room in a desired range. We explained this system in [2], here we study the whole process and with more details. Figure 9 shows the components of the system and its environment. This system includes a sensor, a heating and cooling unit (hc_unit), and a controller. The controller receives sensor data from the sensor and transmits the command activatec, activateh or switchoff to the hc_unit to respectively activate the cooling or heating process, or switch off the process. Assume that there is a window inside the room and the outside weather blows inside when the window is open. The temperature of the room is slowly changed whether the outside weather is colder or warmer than the current temperature value (i.e., uncertain environment). The controller activates the heating/cooling process based on the sensed temperature value. The physical process is temperature regulation, and the desired state is a specific range for the temperature.

6.1. TCS Timed Rebeca model

There are two differences in the modeling of the TCS case study as compared to the PCS system. First, the sensor in the TCS has periodic function for sensing temperature, whereas the sensors of the cylinders in the PCS are triggered by movements and transmit updates to the controllers. Secondly, the environment of the PCS is isolated, with no uncertain behaviors that affect the system functionality, while in the TCS the changing and uncertain

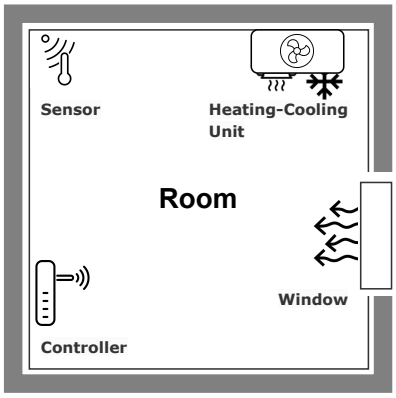

Figure 9: Temperature Control System (TCS) of the room and its environment [2]. The outside air affects the temperature and the sensor reports the temperature changes to the controller. The Heating-Cooling Unit is controlled by the controller to regulate the temperature of the room in a desired range.

temperature value of the environment can impact the functionality of the system.

We demonstrate the Timed Rebeca model (augmented with attacks) of the TCS case study in Listing 4. We assume that the temperature of the room is within the desired range at the beginning (i.e., the value 22 which is between 21 and 23) (line 1). We define three reactive classes Controller, Sensor, and HC_Unit to model the system components and one reactive class Room to model the environment. The state variable sensedValue in the controller stores the sensor data sent by the sensor, and the state variables heating_active and cooling_active respectively show whether the cooling process or the heating process is activated (line 5). In order to regulate the temperature, a message including the value of the temperature (line 38) is sent by the reactive class Sensor to the reactive class Controller. The controller decides to send switchoff to the hc-unit if the temperature value is within the desired range (line 10). If the temperature value is higher/lower than the desired range, the controller produces an appropriate command, i.e., activatec or activateh, and sends it to HC_Unit (lines 13 and 15). The state variable temperature in the reactive class Room shows the value of the temperature of the room. The state variable outside_air_blowing with the non-deterministic assignment shows the outside air blowing inside when the window is open (line 27). The temperature of the room is slowly affected by

outside air blowing, whether the outside weather is colder or warmer than the current temperature value of the room i.e. we do not have a sudden temperature change (line 23). The message server status updates value of the temperature after 10 units of time (line 29) and provides the updated value of the temperature as a response to the request of the sensor (line 30). The message server regulate gets a regulation value from the controller and sets the process to increase and decrease the temperature value (line 31). The Sensor periodically senses the temperature after 2 units of time and sends the updated value to the controller (line 37). The main block includes the declarations of all reactive classes defined in the model (line 51 to 56).

6.2. TCS attack modeling in Timed Rebeca

The goal of attacks in this system is to change the temperature out of the desired range or cause damage to the physical infrastructure (i.e., the heating and cooling unit). We assume that the attacker can compromise the controller to tamper the commands issued by the controller (e.g., code injection attack), and alter the sensor to send false sensor data. These attacks can be modeled by defining Compromised Sensor and Compromised Controller as shown in Figure 10. The coordinated attacks are also modeled by combining both the compromised versions of the sensor and the controller. Similar to the approach in modeling compromised version of the components in the PCS case study, we define Sensor(compromised, compTime, msg) (line 35) and Controller(compromised, compTime, msg) (line 6) as shown in Listing 4, and count the number of false sensor data and faulty control commands as the number of attacks by changing the values of the parameters.

In the attack modeling, Compromised Sensor sends false sensor data to the controller by transmitting values ranging from 19 to 25 when sensing the temperature value (Listing 4, line 2). Also, Compromised Controller sends faulty commands activatec, activateh or switchoff to HC_Unit (Listing 4, line 16 to 19). The complete Timed Rebeca code including the defined attacks is available on GitHub [28].

6.3. TCS safety properties

Listing 5 shows the safety properties that are defined for preventing any unsafe activation of cooling or heating processes in HC_Unit. The Timed Rebeca model without attacks for the TCS case study satisfies the safety properties safety_prop1, safety_prop2 and safety_prop3 as listed below.

```
env int desiredValue = 22; // initial value for the desired temperature
env boolean sComp = false;  env int sComp_time = 0; env int smaliciousMsg = 0; ...
reactiveclass Controller(5) {
  knownrebecs { HC_Unit hc_unit; }
  statevars { int sensedValue;  boolean heating_active;  boolean cooling_active; }
  Controller(boolean compromised, int compTime, int msg){
    heating_active = false; cooling_active = false; sensedValue = desiredValue;
    if (compromised) { self.compromise(msg) after(compTime);}}
  msgsrv getsense(int temp) {  sensedValue = temp;
    if (temp <= 23 && temp >= 21) {
      if (heating_active == true || cooling_active == true) {
        hc_unit.switchoff(); heating_active = false; cooling_active = false; }
    } else if (21 > temp) {
      if (heating_active == false) { hc_unit.activateh();  heating_active = true; }
    } else if (23 < temp) {... } }
  msgsrv compromise(int msg){
    if(msg == 0) {hc_unit.switchoff();
    } else if(msg == 1) { hc_unit.activateh();} else if(msg == -1) {...}
}
}
reactiveclass Room(5) {//a temperature is affected by outside air blowing
  knownrebecs { Sensor sensor; }
  statevars { int temperature;  int outside_air_blowing; int regulation; }
  Room() {
    temperature = 22;  regulation = 0; outside_air_blowing = 0; self.status();}
  msgsrv status() { //enviroment effects the temp slowly, in each 10 units of time
    outside_air_blowing = ? (1, 0, -1);
    temperature = temperature - outside_air_blowing + regulation; //update temp
    self.status() after(10); }
  msgsrv reqsensor() { sensor.getTemp(temperature); }
  msgsrv regulate(int v) { regulation = v; }// regulate temp
}
reactiveclass Sensor(5) {
  knownrebecs { Room room; Controller controller; }
  Sensor(boolean compromised, int compTime, int msg){
    if (compromised) {self.getTemp(msg) after(compTime);} self.sense();}
  msgsrv sense() { room.reqsensor(); self.sense() after(2); }
  msgsrv getTemp(int temp) {  controller.getsense(temp); }
}
reactiveclass HC_Unit(5) {
  knownrebecs { Room room; }
  statevars { boolean heater_on, cooler_on; int regValue; }
  HC_Unit() {
    heater_on = false; cooler_on = false; regValue = 0; self.regulateTemp();
  }
  msgsrv activateh() { regValue = 1; heater_on = true; }
  msgsrv activatec() { regValue = -1; cooler_on = true; }
  msgsrv switchoff() { regValue = 0; cooler_on = false; heater_on = false; }
  msgsrv regulateTemp() { room.regulate(regValue); self.regulateTemp() after(5); }
}
main {
  Room room(sensor): ();
  Controller controller(hc_unit):(cComp,cComp_time,cmaliciousMsg);
  Sensor sensor(room,controller):(sComp,sComp_time,smaliciousMsg);
  HC_Unit hc_unit(room): ();
}
```

Listing 4: Timed Rebeca model for the TCS case study augmented with attacks.

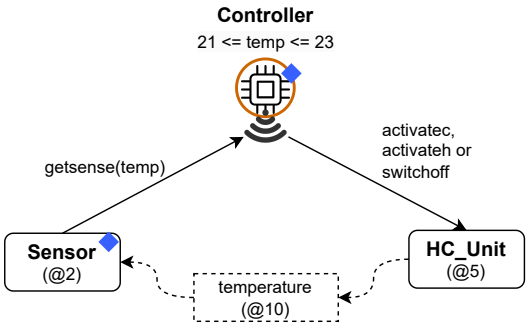

Figure 10: The sensor periodically senses the temperature every 2 units of time and sends the updated value with the message getsense to the controller. The controller transmits an appropriate command to regulate the temperature value between 21 and 23. The HC_Unit changes the heating/cooling process after 5 units of time. The temperature of the room is affected by the environment after 10 units of time. The attack points for performing attack scenarios are the sensor and the controller (depicted with blue diamonds).

The property safety_prop1 ensures that the heating process is not activated when the sensed temperature value is higher than 23 degrees (i.e., sensedValue_over). Indeed, if the sensed temperature is above the desired range, the controller must not send the command activateh to the HC_Unit. Similarly, the property safety_prop2 ensures that the cooling process is not activated when the sensed temperature value is lower that 21 degrees (i.e., sensedValue_under). Finally, the safety property safety_prop3 is violated if the temperature goes outside of the desired range due to any event.

```
property {
    define {
        sensedValue_over = controller.sensedValue > 23;
        sensedValue_under = controller.sensedValue < 21;
        unit_heating = hc_unit.heater_on == true;
        unit_cooling = hc_unit.cooler_on == true;
    }
    Assertion {
        safety_prop1: !(sensedValue_over && !unit_heating);
        safety_prop2: !(sensedValue_under && !unit_cooling);
        safety_prop3: !(sensedValue_under || sensedValue_over);
    }
}
```

Listing 5: The safety properties for the TCS case study.

6.4. TCS security analysis

Table 5 shows the results of the security analysis based on the model checker outputs after performing attacks. We model 4301 number of attacks during a predefined system execution period of 25 seconds using the compromised sensor and controller. This duration corresponds to the completion time of one cycle for sensing, sending commands, and actuation that is modeled for the control system. Among the total number of 269 attacks (182 false sensor data and 87 faulty control commands), 270 attack scenarios successfully violate the safety properties. We also perform combined attack scenarios where the compromised sensor and controller are performed pairwise where each compromised component can not separately violate safety properties. We found 74 successful violations of the safety properties out of the 4032 combined attack scenarios.

As shown in Table 5, Compromised Sensor violates the safety properties with higher numbers compared to Compromised Controller (highlighted in pink). It indicates that attacks on the sensor can be more successful. We also observe that the property safety_prop3 has the highest number of violations (highlighted in blue).

Table 5: The failure of properties for the compromised sensor and controller

	safety_prop1	safety_prop2	safety_prop3	total_successful_attacks
Compromised Controller	9	7	52	68
Compromised Sensor	33	50	45	128
CompSensor+CompController	16	27	31	74
total_fails_prop	58	84	128	270

6.5. TCS Tiny Digital Twin and Monitoring

In this case study, the actions getsense, activateh and switchoff are observable in the system behavior from the controller point of view, while actions gettemp, sense, regulate, reqsensor, status and regulateTemp are non-observable (i.e., silent transitions). The Tiny Digital Twin with 125 states and 154 transitions is created by abstracting the states space which has 799 states and 1440 transitions.

We show a subset of the state transitions of the Tiny Digital Twin to explain the system behavior at different states (see Figure 11). In the Tiny Digital Twin of the temperature control system, we see branching states (e.g., S32 and S22) that present different control flow paths, where Controller decides to activate/switch off HC_Unit regarding the received sensor data. Also,

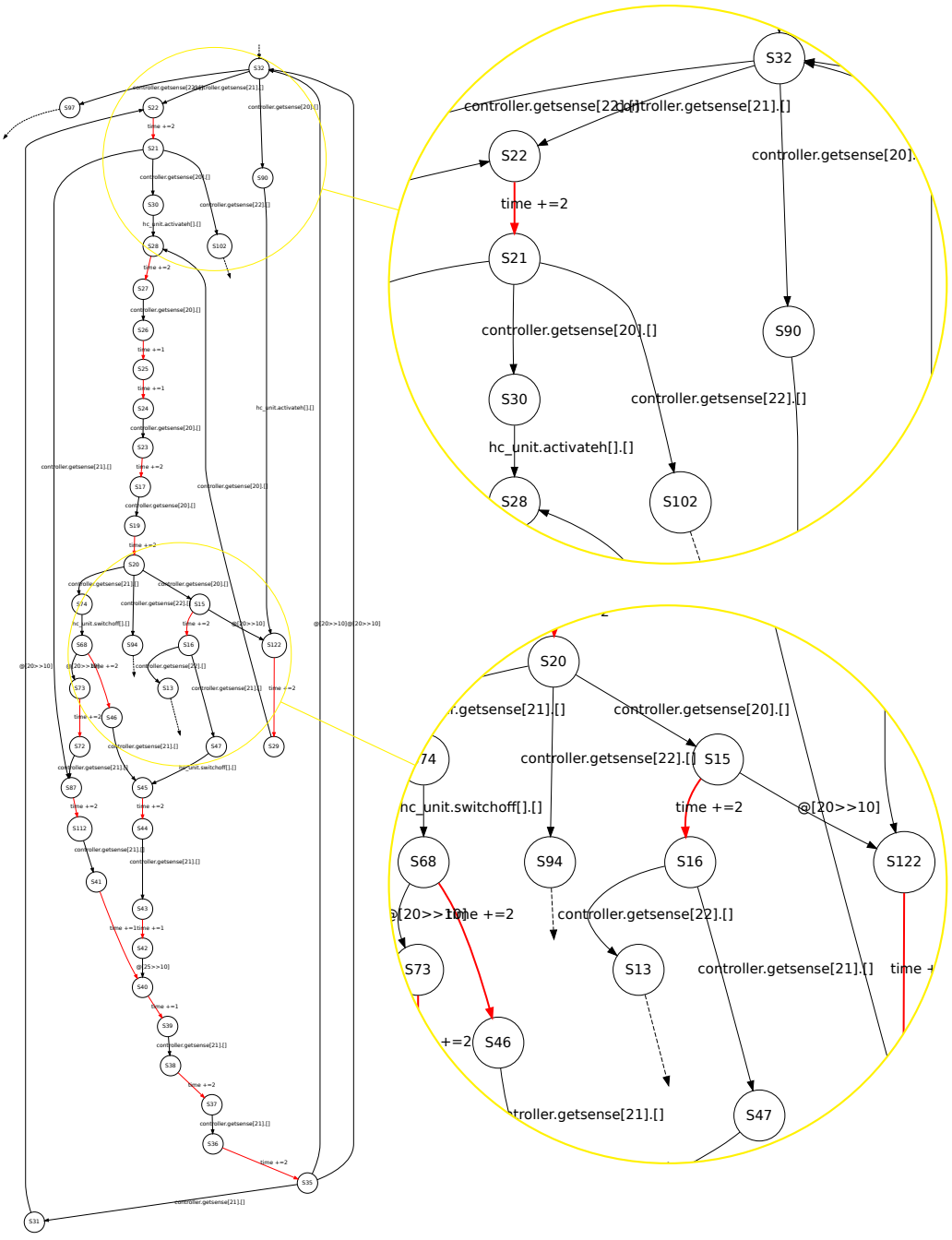

Figure 11: A subset of the state transitions in the Tiny Digital Twin of the TCS case study. It includes branching states and cycles.

there are some cycles of sensor data transmission and control commands, where the same sensor data and control commands are repeated (e.g., a circle consisting of the states S28 to S20, S15, S122, S29 and back to S28). The Tiny Digital Twin is used within Monitor to detect attacks on sensor data and control commands. To prevent damage to the system, the monitor drops control commands that are not consistent with the state transitions in the Tiny Digital Twin. We evaluate the detection capability of Monitor by simulating attacks in Lingua Franca code.

6.6. TCS Lingua Franca

The LF code implements all components of the TCS case study (Listing 6). Diagram 12 shows the connectivity between the components that are defined as reactors. The input port getsense in the reactor Controller (line 3) is defined to get a sensor value, and three output ports activateh, activatec, and switchoff (lines 4-5) are defined to send values as commands to the HC_Unit. We set the value of activateh to 1 to trigger the heating (line 15), the value of activatec to -1 to trigger the cooling (line 17) and the value of switchoff to 0 to switch off the HC_Unit (line 11).

We use a timer to periodically invoke the reactions and model the periodic events (similar to those message servers in Timed Rebeca that send messages to themselves with after). Here, the reaction start in the reactors Room, Sensor and HC_Unit are defined for updating (line 26), sensing (line 36) and regulation (line 46) the temperature which are triggered periodically. For example, the timer start$(0, 10\ \text{sec})$ indicates that updating temperature value is triggered at the start of execution and then it repeats at intervals of 10 seconds (see line 36).

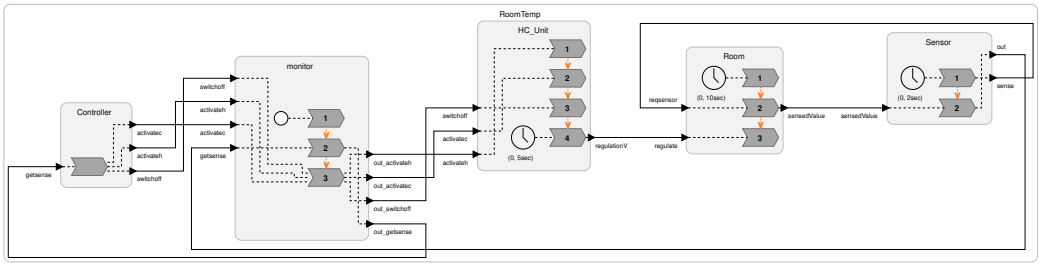

Figure 12: The diagram of the reactors in LF code for TCS case study including monitor reactor.

```
target Cpp {fast: false, threads: 1};
reactor Controller {
    input getsense:int;  //input and output ports
    output activateh:int; // activates hc-unit by sending a value to output
    output activatec:int;  output switchoff:int;
    state heating_active:bool(false); state cooling_active:bool(false);
    reaction(getsense) -> activatec, activateh, switchoff {=
     activatec, activateh, switchoff {=
     if(*getsense.get() <= 23 && *getsense.get() >= 21){
            if(heating_active == true || cooling_active == true){
                switchoff.set(0);
                heating_active = false; cooling_active = false;
            }
        } else if(*getsense.get() < 21) {
            if(heating_active == false){ activateh.set(1); heating_active = true;}
        } else if(*getsense.get() > 23)        {
           if(cooling_active == false){ activatec.set(-1); cooling_active = true;}
            }
        }
    =}
}
reactor Room {
    input regulate:int; input reqsensor:int;
    output sensedValue:int; state temperature:int(22);
    state cold_air_blowing:int(0); state regulation:int(0);
    timer start(0, 10 sec); // triggers room to update temp in each 10 sec
    reaction(start) {=
        cold_air_blowing = rand() % 3 + (-1);
        temperature = temperature - cold_air_blowing + regulation;
     =}
    reaction(reqsensor) -> sensedValue {=  sensedValue.set(temperature); =}
    reaction(regulate) {=  regulation = *regulate.get(); =}
}
reactor Sensor {
    input sensedValue:int; output sense:int; output out:int;
    timer start(0, 2 sec); // triggers sensor in each 2 sec
    reaction(start) -> sense{= sense.set(1); =}
    reaction(sensedValue) -> out {= out.set(sensedValue.get());  =}
}
reactor HC_Unit {
    input activateh:int; input activatec:int; input switchoff:int;
    output regulationV:int; state regValue:int(0);
    reaction(activateh) {=  regValue = 1; =}
    reaction(activatec) {=  regValue = -1; =}
    reaction(switchoff) {=  regValue = 0; =}
    timer start(0, 5 sec); // regulate temperature in each 5 sec
    reaction(start) -> regulationV {=  regulationV.set(regValue);  =}
}
main reactor RoomTemp {
        room = new Room(); sensor = new Sensor(); unit = new HC_Unit();
        controller = new Controller();
        room.sensedValue -> sensor.sensedValue;
        sensor.sense -> room.reqsensor; sensor.out -> controller.getsense;
        ...
}
```

Listing 6: LF code for the TCS case study.

6.7. Detection capability of the monitor in TCS

We develop Monitor in LF where it keeps the Tiny Digital Twin to track the behavior of the system and detect attacks. Similar to the implementation of the attacks in LF code for PCS, we develop Compromised Sensor and Compromised Controller to perform attack scenarios listed in Table 5. We simulate 269 false sensor data and faulty control commands, and also 74 combined attack scenarios. As shown in Table 6, out of the 270 attack scenarios (combined and single successful attacks), 119 attacks are not directly detected by the monitor, therefore the detection rate is about 55 percent. The undetected attacks are related to the compromise of the sensor where the sensor sends the false sensor data that matches to the data on the branches in the Tiny Digital Twin.

Table 6: The detection capability of the monitor in TCS case study

attack types	total_successful_attacks	detection capability
Compromised Controller	68	68
Compromised Sensor	128	52
CompSensor+CompController	74	31
total_detection_rate	270	151 (%55)

In the following, we describe a few examples of attack scenarios using the Tiny Digital Twin depicted in Figure 11 and explain how the Monitor detects the attacks. Assume that the system is in the state S27 and Compromised Sensor sends 21 as the temperature value to the Controller whereas the actual temperature that is sensed by the sensor is 20. Upon receiving this false sensor data, Monitor produces an alarm and terminates the monitoring process because the data sent by Compromised Sensor is not matched with the data on the transition of the Tiny Digital Twin. As another example, assume the system is in the state S90 and the Compromised Controller sends activatec as a faulty control command. The Monitor drops the command because it is not consistent with the outgoing transition of S90. A coordinated attack can occur when Compromised Sensor sends false sensor data and Compromised Controller alters the command to overwrite the controller decision caused by the false sensor data.

From states S32, S21, S20, S35 and S16 there are multiple outgoing transitions. For instance, assume that 21 is sensed as the temperature value in S32 but Compromised Sensor sends the value 24. According to the Tiny Digital Twin of the case study, the value for the next states can be either 20 (S90), 21 (S22), or 22 (S97), therefore Monitor detects the false sensor data.

Note that the controller should in principle sends activatec to activate the cooling process by sensing 24. But this is where in modeling the behavior of the environment, in the Timed Rebeca model, we do not model any jumps in the temperature from 21 to 24. Therefore, this is captured as an unexpected behavior. As another example, assume that the value 22 is sensed as the temperature value in S32 but the Compromised Sensor sends a sensed value 21 or 20. In this case, Monitor can not detect the false sensor data. We are able to use meta-rules to check if the paths between turning on the heating (or cooling) unit(s) are taken too quickly, or if any of these processes stay turned on for a time longer than expected. This is one of the ways in which we will continue our research.

7. Secure Water Treatment System (SWaT)

The SWaT testbed [29] is a scaled-down version of an industrial water treatment system. This testbed is used for several research and training purposes in the iTrust research center [29]. We present the details of SWaT architecture and its Timed Rebeca model (adapted from [1]). We explain the definition of the priority for the message servers in the Timed Rebeca, and then provide the Tiny Digital Twin and the attack detection using the monitor.

The water treatment process in the SWaT system consists of three stages as shown in Figure 13. These stages include supplying raw water into the system, Ultra-Filtration (UF) and Reverse Osmosis (RO). In each stage, there is a PLC responsible for controlling a water tank. The PLC is directly connected to some actuators (i.e., valves or pumps) through a local network. A simple password-based authentication is the only mechanism employed to control access to the network, which makes the SWaT system vulnerable to eavesdropping or packet injection attacks [30].

At any stage during the execution of the water treatment process, each pump can be in on or off state, and respectively each valve can be in one of the two states open or close. Also, three states are considered for the big tanks (i.e., Tank_1 and Tank_2): Low(l), Medium(m), and High(h), and two states for the small tank (Tank_3): Low(l) and High(h). During the system operation, whenever the water level of a tank changes to h, the associated sensor reports the change to the responsible PLC. That PLC will close the valve or turn off the pump that is pouring water into the tank. Also, the PLC may open a valve, turn a pump on, or send open/on requests to other

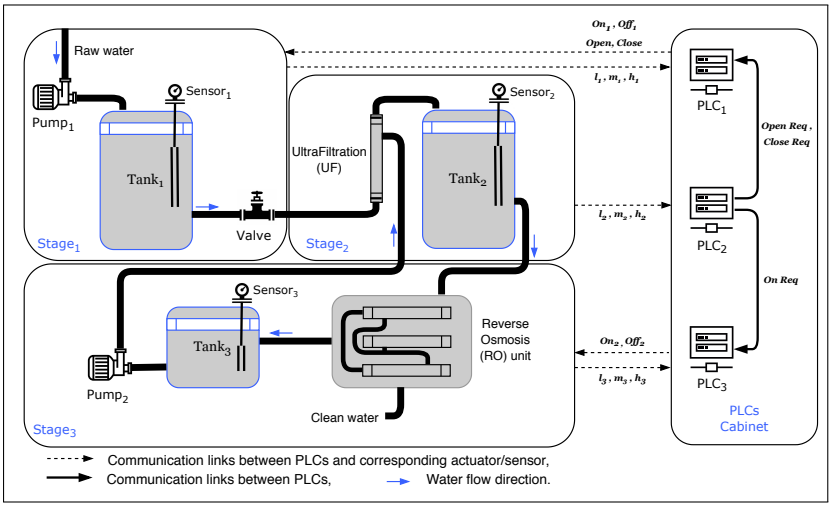

Figure 13: An abstract architecture of the SWaT system (adapted from [31]).

PLCs when the water level in the tank is either l or m. The PLC_1, PLC_2 and PLC_3 are configured to interact with each other to manage the SWaT system.

7.1. SWaT Timed Rebeca model

The PLCs communicate with each other through a separate protected network. For example, the open_Req/close_Req or the on_Req messages passed in the secured channel between the PLCs may not be the target of any attacker. However, the messages (l, m, and h) which are transmitted from the sensors to the PLCs may be tampered by an attacker to affect the decisions made by the PLCs. The blue points represent the components that may behave maliciously. Typically, the malicious behavior of the component leads to a faulty data transmission. For instance, whenever a pump is compromised, it may push the water to the connected tank once it receives the command off from the corresponding PLC.

In the SWaT Timed Rebeca model, we assume that the water level in each tank is low in the initial state. Also, the water treatment process begins by pumping raw water to Tank_1 and it ends when the cleaned water flows out of Tank_2. During the process execution, each sensor sends water level information to the corresponding PLC periodically.

In the following, we provide a detailed explanation of the Rebeca model (see Listing 7) developed for the SWaT system. The complete model is

available in [32]. The main block includes the declarations of all rebecs together with an attacker rebec (line 45 to 50). In addition to the main block, the Rebeca model includes the reactive classes defining the behavior of the SWaT actors. For example, the PLC_1 reactive class has two known rebecs which are instances of reactive classes Pump_1 and Valve. The reactive class PLC_1 includes a boolean state variable openReqPlc2 whose value indicates whether a water request is received from PLC_2 or not (i.e. request for opening the valve). This variable is initialized to false in the constructor of PLC_1.

Two boolean state variables pump1On and valveOpen indicate the current status of Pump_1 and Valve respectively. The definition of PLC_1 includes three message servers i.e., plc1_getsense, plc1_openReq and plc1_closeReq (see listing 7 line 5 to 14). The message server plc1_getsense processes the sensor data and issues commands on or off to Pump_1 and open or close to Valve accordingly. The message servers plc1_openReq and plc1_closeReq are activated once a message is received from PLC_2. The definition of priority depends on the programming language and the techniques in the controllers such as PLCs. For example, in ladder logic programming for PLCs, the priority of incoming messages can be defined using timers or counters. We can use @priority before the definition of message servers in a reactive class to specify the sequence in which messages received by the actor are executed. In this model, the message server plc1_closeReq has higher priority than the message servers plc1_openReq and plc1_getsense. In order to have correct functionally when messages arrive at the same time, we set the priority of the message server plc1_closeReq to 1 using @priority(1). The reactive class Tank_1 contains two message servers, tank1_waterIncrease and tank1_waterDecrease, which are used to change the level of water in the tank. We model in a way where the messages corresponding to water decrease are executed before the messages related to water increase. The reactive class Pump_1 includes four message servers on, off and KeepOnpumping (lines 31 to 50). The message servers on and off update the value of the state variable On based on the commands received from PLC_1. The message server KeepOnpumping calls waterIncrease and increases the level of water for one level in the tank. This continues until the message server off receives the turn off message. The main block includes the declarations of the reactive classes in the model including the priority for each reactive class.

```
env int chl = 1;  env int malMsg = 0;  env int attackTime = 0;  ... //env variables
reactiveclass PLC_1(5){  knownrebecs{ Pump_1 pump1; Valve valve;}
    statevars{ boolean openReqPlc2, pump1On, valveOpen; int waterLevelTank1;}
    PLC_1(){openReqPlc2 = false; sensedTank1WaterLevel = 1; pump1On = false; ...}
    @priority(3) msgsrv plc1_getsense(int waterLevel){
        if (waterLevel == 1 && pump1On == false) {
            pump1.pump1_on(); pump1On = true;
        } else if (waterLevel == 2) {
            if(!valveOpen  && openReqPlc2) {
                valve.valve_open(); valveOpen = true;
            } else if(valveOpen  && openReqPlc2) {valve.valve_keepOnWaterFlow(); } ...
        } else {... }
    @priority(2) msgsrv plc1_openReq(){openReqPlc2 = true;}
    @priority(1) msgsrv plc1_closeReq(){
        openReqPlc2 = false; valve.valve_close(); valveOpen = false; }
}
reactiveclass PLC_2(5){...} reactiveclass PLC_2(5){...}
reactiveclass Tank_1(10){ knownrebecs{ sensorTank1 sensor; ...}
    statevars{ boolean underFlow,low,medium,high,overFlow;}
    Tank_1(){ underFlow = false; overFlow = false; low = true; ...}
    msgsrv tank1_status(){ if (underFlow){sensor.sensor1_reportStatus(0);} else {...}
    @priority(1) msgsrv tank1_waterIncrease(){ ... //changes water level status
        if (low == true) { medium = true; low = false; high = false;
        } else if (medium == true) {...}}
    @priority(2) msgsrv tank1_waterDecrease(){...
        if (medium){ tank2.tank2_waterIncrease();}}
}
reactiveclass Tank_2(10){...}
reactiveclass Tank_3(10){... msgsrv tank3_waterDecrease(){
                if (high) {... tank2.tank2_waterIncrease(); } else if (low) {...}}
reactiveclass Pump_1(10){ knownrebecs{ Tank_1 tank1;}
    statevars{ boolean pump1_on;}
    Pump1(boolean compromised, int compTime, int msg){ pump1_on = false;
        if (compromised == true) {
            if(msg == 1){self.pump1_on();} else {self.pump1_off();}}
    msgsrv pump1_on(){
        pump1_on = true; tank1.tank1_waterIncrease(); }
    @priority(2) msgsrv pump1_keepOnpumping(){
        if (pump1_on) {tank1.tank1_waterIncrease(); }}
    @priority(1) msgsrv pump1_off(){...}
}
reactiveclass Pump_2(10){...} reactiveclass Valve(10){...}
reactiveclass sensorTank1(10){...} reactiveclass sensorTank2(10){...}
reactiveclass sensorTank3(10){...} reactiveclass reverseOsmosisUnit(5){...}
reactiveclass Attacker(3){ knownrebecs{ PLC_1 plc1; PLC_2 plc2; ...}
    Attacker(boolean inj, int channel, int maliciousMsg, int attackTime){
        if (inj && chl == 1) { self.channelPlc1S(maliciousMsg, attackTime);
        } else if (inj && chl == 2) {...}}
    msgsrv channelPlc1S(int msg, int attackTime){
        plc1.plc1_getsense(msg) after(attackTime);} ... } //message servers
main{
    @priority(2) PLC_1 plc1(pump1, valve):();
    @priority(3) sensorTank1 sensor1(tank1, plc1):(s1Comp,s1Comp_time,s1malMsg);
    ...
    Attacker attacker(plc1,plc2,plc3,pump1,pump2,valve):(inj_attk,chl,malMsg,attTime)}
```

Listing 7: An abstract version of the SWaT system Timed Rebeca model augmented with attacks.

7.2. SWaT attack modeling in Timed Rebeca

In this experiment we use model checking to detect the undesirable events that might happen while attackers tamper the channels (e.g., by injecting packets) and compromise sensors/actuators by altering their functionality.

In the Timed Rebeca model, we model compromised actors using parameters that are passed to them (see Listing 7). For example, the reactive class Pump_1 includes a variable compromised that can be set to change the status of the component to be compromised or not compromised (line 33). If this variable is set to be compromised then although the pump receives a message to turn its status to on, it turns it to off. For changing the variable compromised at different times we use the parameter compTime. Similar to the compromised mode of Pump_1, whenever the value of the input parameter compromised is true for Valve, then both message servers open and close behave maliciously (for example the message server open changes the state variable Open of Value to false). In addition to the reactive classes that define the normal and compromised behavior of SWaT components, the Rebeca model includes a reactive class named Attacker (line 45) that models the behavior of potential attackers targeting channels to inject messages.

7.3. SWaT safety properties

The goal of attacks on the SWaT system is to cause overflow or underflow in one of the tanks. An overflow may harm some of the critical units such as the UF or RO and lead to flow out unclean water, and an underflow may damage a valve or a pump. Accordingly, the safety properties presented in Listing 8 are considered to be verified on the Timed Rebeca model of SWaT system. These properties ensure that each tank has no overflow or underflow.

7.4. SWaT security analysis

The outcome of the security analysis includes the attack scenarios which lead the system to the property violation. We write a Python script to put different values for the input parameters of the attacker and the compromised components, and verify the model for each combination.

As described in the previous work [1], we modeled 105 communication attacks and 84 attacks on components, and also the combination of these attacks (resulting in 8820 attack scenarios). Totally, out of all above possible attack scenarios 29 cases successfully violate the system security.

The analysis results in Table 7 indicate Combined attacks, such collaborative attacks can be easily detected. For example assume that the system

```
property {
    define {
        t1_overFlow = tank1.overFlow; t1_underFlow = tank1.underFlow;
        t2_overFlow = tank2.overFlow; t2_underFlow = tank2.underFlow;
        t3_overFlow = tank3.overFlow; t3_underFlow = tank3.underFlow;
    }
    Assertion {
        safety_tank1_overflow: !(t1_overFlow);
        safety_tank1_underflow: !(t1_underFlow);
        safety_tank2_overflow: !(t2_overFlow);
        safety_tank2_underflow: !(t2_underFlow);
        safety_tank3_overflow: !(t3_overFlow);
        safety_tank3_underflow: !(t3_underFlow);
    }
}
```

Listing 8: The safety properties for the SWaT system.

is executing and an attacker injects message Open Valve into the communication link between PLC_1 and Valve, and at the same time another attacker compromises Pump_1 to be turned off, then Tank_1 will underflow (line 1 in Table 7). As another example, Sensor_2 is compromised and a malicious message of high water level for Tank_3 is injected into the channel between Sensor_3 and PLC_3, then Sensor_3 will underflow (line 3 in Table 7).

Note that the scenarios presented in Table 7 are those in which the single attacks (message injection or the compromised component) do not cause a security failure separately, but the combination leads to the security violation. If we assume that the system is robust against the single attack scenarios, the system may still be vulnerable against the collaborative attacks.

Table 7: The failure of properties in Combined Attack.

#	Property	Injected message	Communication channel	Compromised component	Malicious behavior
1	safety_tank1_underflow	Open Valve	(PLC_1 to Valve)	Pump_1	(Turn Off)
2	safety_tank3_underflow	Water level inTank_2 is medium	(Sensor_2 to PLC_2)	Sensor_3	(Water level in Tank_3 is high)
3	safety_tank3_underflow	Water level in Tank_3 is high	(Sensor_3 to PLC_3)	Sensor_2	(Water level in Tank_2 is medium)

7.5. SWaT Tiny Digital Twin and Monitoring

The actions getsense for the sensor data, on/off/keepOnpumping for the pumps and open/close for the valve are observable in the SWaT system behavior from the controller point of view. We use *ltsconvert* tool to create the Tiny Digital Twin with 187 states and 303 transitions, which has abstracted the states space with 543 states and 728 transitions.

We show a subset of the state transitions of the Tiny Digital Twin to explain the system behavior at different states (see Figure 14). In the Tiny Digital Twin of the SWaT system, we observe that PLCs take decision to turn on/off pumps and open/close valve based on the received sensor data. As shown in the diagram, the water level for each tank is sent by the corresponding sensors, and the water level in Tank_1, Tank_2 and Tank_3 is low at the beginning (the outgoing transitions from S0 to S207). The order of the events for sensor data can be different which are indicated in the state transitions. Regarding the status of the system, PLC_1 turns on Pump_1 and increases the water level in Tank_1 (S207). PLC_2 sends openReq to ask PLC_1 to open Valve if the water level in Tank_1 is medium (S233). PLC_3 turns off Pump_2 because the water level in Tank_3 is low (S136). The water level in Tank_1 reaches medium after 10 units of time and causes PLC_1 opens Valve while Pump_1 is on and increases the water level in Tank_1 (S119). The sensor data are sent by the sensors to the PLCs and report the water level in each tank (S37).

The Tiny Digital Twin is used within Monitor to detect attacks that compromise the actions getsense for the sensor data, on/off/keepOnpumping for the pumps and open/close for the valve. We evaluate the detection capability of Monitor by simulating attacks in Lingua Franca.

7.6. SWaT Lingua Franca

There are two restrictions when connecting reactors and writing reactions in LF code. Firstly, each input port in a reactor can have at most one incoming connection, while each output port can have multiple outgoing connections. Secondly, all reactions within a reactor are executed in the order they are presented.

In the Timed Rebeca model of SWaT case study, we have priority for the execution of the messages by using the @priority notation for message servers such as tank1_waterIncrease and tank1_waterDecrease (see Listing 7 line 25 and 22). Additionally, some message servers execute messages that are sent from different reactive classes, for example, tank2_waterIncrease in Tank_2 execute the messages sent from the reactive classes Tank_1 and Tank_3 (see Listing 7 line 26 and 30). In the mapping between Timed Rebeca and LF, we write the reactions within a reactor with the order defined in the timed Rebeca model based on the priority. For example, we present the reaction close_Req before the reactions open_Req and getsense within the reactor PLC1 (see Listing 9 line 6, 7 and 8). Also, those message servers that receive

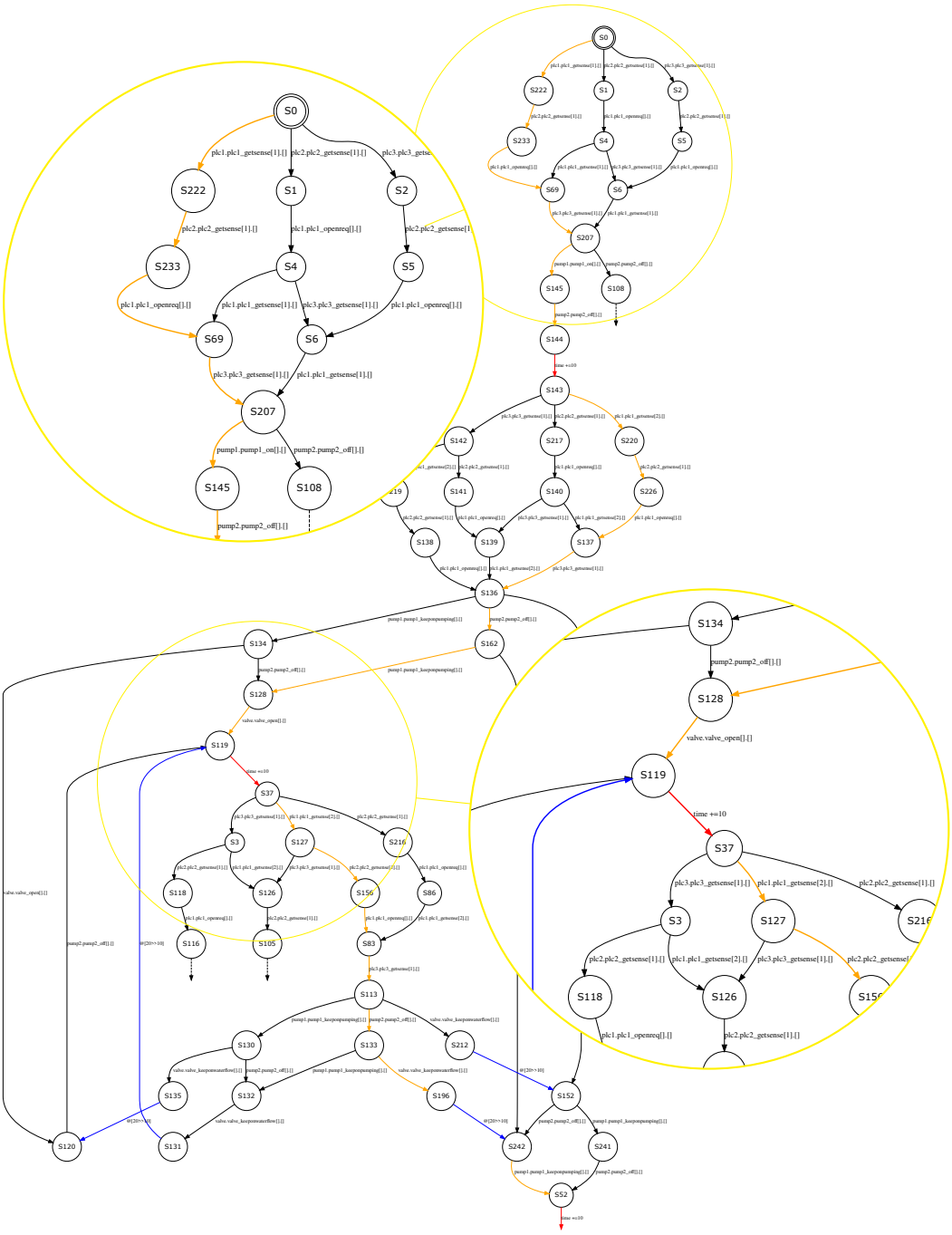

Figure 14: A subset of the state transitions in the Tiny Digital Twin of the SWaT case study.

messages from different reactive classes are defined with the separate input ports in the reactor (see Listing 9 line 28).

7.6.1. Detection capability of the monitor in SWaT

We use Compromised version of the sensors, pumps and valve, and Attacker component to create false data and faulty commands at different times during execution. Among the attacks that violate the properties (i.e., 29 cases from the security analysis), Monitor directly detects the attacks that involve false sensor data. The attacks on actuators (i.e., pumps and valve) is detected when the sensor data is reported to the Monitor. Therefore, as shown in Table 8, the detection rate of the monitor is about 51 percent in this system.

Table 8: The detection capability of the monitor in SWaT case study

attack types	total_successful_attacks	detection capability
Attacks on Communication	15	7
Attacks on Components	11	6
Combined Attacks	3	2
total_detection_rate	29	15 (%51)

In the following, we explain a combined attack scenario and the detection of the attack by Monitor. In Figure 15, we assume that the system is under a combined attack $\mathsf{CompSensor}_2 + \mathsf{injection}$. At the time of the attack, the water level in Tank_1, Tank_2, and Tank_3 is low, and Sensor_1 and Sensor_3 send the message l to the PLC_1 and PLC_3, respectively. However, (1) Sensor_2 is compromised, and it sends false sensor data m to PLC_2, even though the water level it senses is l. Based on the received false sensor data, (2) PLC_2 sends an on_Req command to PLC_3. At the same time, (3) Attacker injects false sensor data h into the communication channel between Sensor_3 and PLC_3. As a result, when PLC_3 receives the on_Req command, (4) it decides to issue a command to turn on Pump_2 because the water level in Tank_3 is high as reported. (5) The faulty command causes Pump_2 to turn on, which decreases the water level in Tank_3 even though the actual water level is low. It is important to note that the compromised sensor and the command injection by the attacker cannot violate the system separately because PLC_3 needs both the on_Req and the the sensor data indicating h to turn on Pump_2.

According to the Tiny Digital Twin of the SWaT system in Figure 14, the Monitor can detect false sensor data transmissions and injections in the combined attack scenario. As highlighted in the state transition diagram, it

```
target Cpp {fast: false, threads: 1};
import Monitor.lf;
reactor PLC1 {
    input close_Req:int; input getsense:int; input open_Req:int; //input ports
    output pump_on:int;  output pump_off:int; output pump_keepOnpumping:int; ...
    reaction(close_Req) {= openReqPlc2 = false;  valveOpen = false; =}
    reaction(open_Req) {= openReqPlc2 = true; =}
    reaction(getsense) -> pump_on, pump_off, pump_keepOnpumping, ... {=
        if(getsense.is_present() && close_Req.is_present()){ //priority of close_Req
            openReqPlc2 = false; valve_close.set(1); valveOpen = false; }
        if(*getsense.get() == 1 && !pump1On) { pump_on.set(1); pump1On = true;
        } else if(*getsense.get() == 2) { ...
        } else if(*getsense.get() == 3) {... =}
}
reactor PLC2 {...}reactor PLC3 {...}
reactor Tank1 { output reportStatus:int; output waterFlows:int;
    input status:int; input status_V:int; input status_T:int;
    input waterIncrease:int; input waterDecrease:int; state waterLevel:int(1);
    timer start(0, 10 sec);
    reaction(start, status, status_V, status_T) -> reportStatus {=
            reportStatus.set(waterLevel);
        =}
    reaction(waterIncrease) {= waterLevel = waterLevel + 1;=}
    reaction(waterDecrease) -> waterFlows {=
        if(waterLevel == 2){ waterFlows.set(1); // water flows into tank2
        } else {waterLevel = waterLevel - 1;}=}
}
reactor Tank2 {... input waterIncrease_T1:int; input waterIncrease_T3:int;
    reaction(waterIncrease_T1) -> waterFlows{=...=}
    reaction(waterIncrease_T3) -> unit,status_t1,reportStatus {=...=}}
}
reactor Tank3 {...} reactor sensorTank1 {...}
reactor sensorTank2 {...} reactor sensorTank3 {...}
reactor Pump1 {...} reactor Pump2 {...}
reactor reverseUnit {...}
reactor Valve { input open:int; input close:int; input keepOnWaterFlow:int;
    output out:int; output status:int;
    reaction(open) -> out {= out.set(1); =}
    reaction(keepOnWaterFlow) {= out.set(1); =}
    reaction(close) -> status {= status.set(1); =}
}

main reactor SWaT {
    monitor = new monitor();
    Tank1 = new Tank1();
    valve = new Valve();
    PLC1 = new PLC1();
    PLC1.pump_on -> pump1.on;
    PLC1.pump_off -> pump1.off;
    PLC1.pump_keepOnpumping -> pump1.keepOnpumping;
    PLC1.valve_open -> valve.open;
    PLC1.valve_close -> valve.close;
    PLC1.valve_keepOnWaterFlow -> valve.keepOnWaterFlow;
    monitor.out_valve_open -> valve.open;
    ....
}
```

Listing 9: LF code for the SWaT case study.

expects to see the water level reported as low for each tank between state S0 and S144. Monitor observes the sensor data transmitted into the network. If it detects any false sensor data or faulty commands, it will drops them to mitigate any potential system failures.

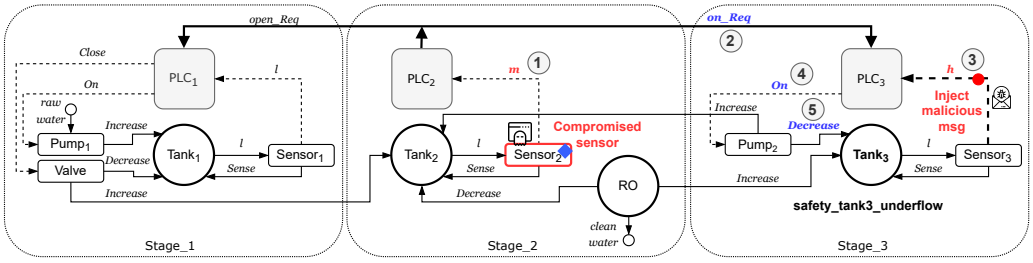

Figure 15: The sequence of events for the combined attack CompSensor_2 + injection where the attack successfully turns on Pump_2 while the water level in Tank_3 is low. The attack causes the system to subsequently generate on_Req and On where violate the safety property for Tank_3.

8. Related Work

There are various techniques for improving the security of industrial control systems. Some of these techniques include deploying security patches, tracking and monitoring critical areas of the system. One approach in the security engineering is to create frameworks that facilitate the design, building, and deployment of secure systems, as well as the analysis and evaluation of the security of existing systems. Following this line of research several modeling and simulation methods are proposed for analyzing the security of cyber-physical systems (CPSs).

Wasicek et al. [33] propose an aspect-oriented technique to model attacks against CPSs. They use Ptolemy [34] as the modeling and simulation framework, and demonstrate the practicality of their technique through modeling four types of attacks on an automotive control system. Taormina et al. [35] propose another simulation-based approach that is implemented in a MATLAB toolbox to analyze the risk of cyber-physical attacks on water distribution systems. In [36, 37], the authors rely on simulation to perform their analyses. They propose a new metric to quantify the impact of attacks on components of the target CPS and perform a cost-benefit analysis on security investments.

In [30], Kang et al. use Alloy to model a scaled-down version of a Secure Water Treatment (SWaT) system behavior and potential attackers. They can discover the undetected attacks which cause safety failure (e.g., water tank overflow). Rocchetto and Tippenhauer [38] use ASLan++ tool for modeling the physical layer interactions and CL-AtSe tool for analyzing the state space. They succeed to find eight attack scenarios on SWaT. Nigam and Talcott [39] use Maude [40] to automate security analysis of the protocols in Industry 4.0 applications. They formalize networked sets of devices and a symbolic intruder model in rewriting logic. Fritz and Zhang [41] consider CPSs as discrete-event systems and model them using a variant of Petri nets. They propose a method based on permutation matrices to detect deception attacks. In particular, they can detect attacks by changing the input and output behavior of the system and analyzing its effect on the system behavior.

The authors in [42] propose monitors expressed as automaton models [43] to detect injection attacks against a system. Their automaton models represent parametric specifications that need to be checked at runtime. Their monitors support event duplication to protect the system against attacks. They validate the approach by implementing the monitors and performing attack examples on a program taken from the FISSC benchmark [44].

9. Conclusion and Future Work

The CRYSTAL framework provides tools for modeling and testing the resilience of cyber-physical systems against cyberattacks. The case studies presented in the paper provide evidence of the effectiveness of CRYSTAL in detecting different types of attacks. We conduct security tests through model checking at design-time and implements the system to simulate and evaluate the detection system at runtime. The cornerstones of the framework are actor-based modeling of the system, defining attack models, abstracting the model and creating Tiny Digital Twin, and finally developing a monitor to detect cyberattacks.

The monitor may not immediately detect attacks on actuators as it must first receive notice from the sensor data and compare it to the Tiny Digital Twin. The delay in detecting attacks can be problematic in systems where each control command can have a significant impact on the physical process, such as fast-moving arms in robotics. In systems where there is enough time to stop the system after successful attacks, the monitor can effectively detect attacks and mitigate the impact of attacks before the significant damage.

For example in TCS, the temperature is changed slowly over time. When an attacker compromises the heating/cooling process, the monitor has enough time to stop the temperature changes by dropping faulty commands.

As a future direction, we plan to provide a module for writing policies and rules within the monitor to catch abnormal behavior of the system and detect attacks that force the system into undesirable states. By embedding these policies and rules into the monitor, the system can become more resilient to cyberattacks, even those attacks that the monitor may not be able to detect immediately.

Acknowledgment

This research is partly supported by Swedish Foundation for Strategic Research (SSF) via the Serendipity project, and KKS SACSys Synergy project (Safe and Secure Adaptive Collaborative Systems).

References

[1] F. Moradi, S. A. Asadollah, A. Sedaghatbaf, A. Čaušević, M. Sirjani, C. Talcott, An actor-based approach for security analysis of cyber-physical systems, in: International Conference on Formal Methods for Industrial Critical Systems, Springer, 2020, pp. 130–147.

[2] F. Moradi, M. Bagheri, H. Rahmati, H. Yazdi, S. A. Asadollah, M. Sirjani, Monitoring cyber-physical systems using a tiny twin to prevent cyber-attacks, in: International Symposium on Model Checking Software, Springer, 2022, pp. 24–43.

[3] J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (2003) 41–50.

[4] E. A. Lee, M. Sirjani, What good are models?, in: International Conference on Formal Aspects of Component Software, Springer, 2018, pp. 3–31.

[5] M. Sirjani, Power is overrated, go for friendliness! expressiveness, faithfulness, and usability in modeling: the actor experience, in: Principles of Modeling, Springer, 2018, pp. 423–448.

[6] M. Sirjani, L. Provenzano, S. A. Asadollah, M. H. Moghadam, From requirements to verifiable executablemodels using rebeca, in: International Workshop on Automated and verifiable Software sYstem DEvelopment, 2019. URL: `http://www.es.mdh.se/publications/5645-`.

[7] M. Sirjani, E. Khamespanah, On time actors, in: Theory and Practice of Formal Methods, Springer, 2016, pp. 373–392.

[8] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, M. Izadi, Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system, Sci. Comput. Program. 98 (2015) 184–204.

[9] M. Lohstroh, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, A. Sangiovanni-Vincentelli, Reactors: A deterministic model for composable reactive systems, in: Cyber Physical Systems. Model-Based Design, Springer, 2019, pp. 59–85.

[10] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, E. A. Lee, Invited: Actors revisited for time-critical systems, in: DAC, 2019.

[11] Afra, An integrated environment for modeling and verifying Rebeca family designs, [Online; accessed Dec 09, 2022] (2022). URL: `https://rebeca-lang.org/alltools/Afra`.

[12] A. Shostack, Threat modeling: Designing for security, Wiley, 2014.

[13] F. Moradi, B. Pourvatan, S. A. Asadollah, M. Sirjani, Tiny twins for detecting cyber-attacks at runtime using concise rebeca time transition system, Submitted to Journal of Parallel and Distributed Computing (2023).

[14] D. N. Jansen, J. F. Groote, J. J. Keiren, A. Wijs, An O (m log n) algorithm for branching bisimilarity on labelled transition systems, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2020, pp. 3–20.

[15] A. H. Reynisson, M. Sirjani, L. Aceto, M. Cimini, A. Jafari, A. Ingólfsdóttir, S. H. Sigurdarson, Modelling and simulation of asynchronous real-time systems using timed rebeca, Sci. Comput. Program. 89 (2014) 41–68. URL: `https://doi.org/10.1016/j.scico.2014.01.008`. doi:`10.1016/j.scico.2014.01.008`.

[16] M. Sirjani, L. Provenzano, S. A. Asadollah, M. H. Moghadam, M. Saadatmand, Towards a verification-driven iterative development of software for safety-critical cyber-physical systems, Journal of Internet Services and Applications 12 (2021) 2.

[17] M. Sirjani, E. A. Lee, E. Khamespanah, Verification of cyberphysical systems, Mathematics 8 (2020) 1068.

[18] M. Lohstroh, C. Menard, A. Schulz-Rosengarten, M. Weber, J. Castrillon, E. A. Lee, A language for deterministic coordination across multiple timelines, in: 2020 Forum for Specification and Design Languages (FDL), IEEE, 2020, pp. 1–8.

[19] M. Sirjani, M. M. Jaghoori, Ten years of analyzing actors: Rebeca experience, in: Formal Modeling: Actors, Open Systems, Biological Systems, Springer, 2011, pp. 20–56.

[20] M. Lohstroh, C. Menard, S. Bateni, E. A. Lee, Toward a lingua franca for deterministic concurrent systems, ACM Transactions on Embedded Computing Systems (TECS) 20 (2021) 1–27.

[21] M. Sirjani, E. Khamespanah, E. Lee, Model checking software in cyberphysical systems, in: COMPSAC 2020, 2020.

[22] T. A. Henzinger, The theory of hybrid automata, in: Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996, IEEE Computer Society, 1996, pp. 278–292.

[23] J. Giraldo, D. Urbina, A. Cardenas, J. Valente, M. Faisal, J. Ruths, N. O. Tippenhauer, H. Sandberg, R. Candell, A survey of physics-based attack detection in cyber-physical systems, ACM Computing Surveys (CSUR) 51 (2018) 1–36.

[24] S. Choi, J.-H. Yun, S.-K. Kim, A comparison of ics datasets for security research based on attack paths, in: International Conference on Critical Information Infrastructures Security, Springer, 2018.

[25] J.-M. Flaus, Cybersecurity of industrial systems, J. Wiley & Sons, 2019.

[26] Z. Jakovljevic, V. Lesi, M. Pajic, Attacks on distributed sequential control in manufacturing automation, IEEE Transactions on Industrial Informatics 17 (2020) 775–786.

[27] Pneumatic control system case study, `https://github.com/fereidoun-moradi/Reconfigurable-Pneumatic-System`, 2023. [Online; accessed Apr 24, 2023].

[28] Temperature control system case study, `https://github.com/fereidoun-moradi/RoomTemp`, 2023. [Online; accessed Apr 24, 2023].

[29] A. P. Mathur, N. O. Tippenhauer, Swat: a water treatment testbed for research and training on ics security, in: Cyber-physical Systems for Smart Water Networks (CySWater), IEEE, 2016, pp. 31–36.

[30] E. Kang, S. Adepu, D. Jackson, A. P. Mathur, Model-based security analysis of a water treatment system, in: Proceedings of Software Engineering for Smart Cyber-Physical Systems, ACM, 2016, pp. 22–28.

[31] R. Lanotte, M. Merro, A. Munteanu, Runtime enforcement for control system security, in: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF), IEEE, 2020, pp. 246–261.

[32] Rebeca homepage, `http://rebeca-lang.org/allprojects/CRYSTAL`, 2020.

[33] A. Wasicek, P. Derler, E. A. Lee, Aspect-oriented modeling of attacks in automotive cyber-physical systems, in: ACM/EDAC/IEEE Design Automation Conference (DAC), 2014.

[34] J. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, Kluwer Academic Publishers, 2001, p. 527–543.

[35] R. Taormina, S. Galelli, N. O. Tippenhauer, E. Salomons, A. Ostfeld, Characterizing cyber-physical attacks on water distribution systems, Journal of Water Resources Planning and Management (2017).

[36] R. Lanotte, M. Merro, R. Muradore, L. Viganò, A formal approach to cyber-physical attacks, in: IEEE 30th Computer Security Foundations Symposium (CSF), IEEE, 2017, pp. 436–450.

[37] R. Lanotte, M. Merro, A. Munteanu, L. Viganò, A formal approach to physics-based attacks in cyber-physical systems, TOPS 23 (2020) 1–41.

[38] M. Rocchetto, N. O. Tippenhauer, Towards formal security analysis of industrial control systems, in: ACM Asia Conference on Computer and Communications Security, ACM, 2017, pp. 114–126.

[39] V. Nigam, C. Talcott, Formal security verification of industry 4.0 applications, in: 24th IEEE International Conference on Emerging Technologies and Factory Automation, 2019.

[40] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude-A High-Performance Logical Framework: How

to Specify, Program, and Verify Systems in Rewriting Logic, volume 4350, Springer, 2007.

[41] R. Fritz, P. Zhang, Modeling and detection of cyber attacks on discrete event systems, IFAC-PapersOnLine 51 (2018) 285–290.

[42] A. Kassem, Y. Falcone, Detecting fault injection attacks with runtime verification, in: Proceedings of the 3rd ACM Workshop on Software Protection, 2019, pp. 65–76.

[43] H. Barringer, Y. Falcone, K. Havelund, G. Reger, D. Rydeheard, Quantified event automata: Towards expressive and efficient runtime monitors, in: International Symposium on Formal Methods, Springer, 2012, pp. 68–84.

[44] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, P. de Choudens, Fissc: A fault injection and simulation secure collection, in: International Conference on Computer Safety, Reliability, and Security, Springer, 2016, pp. 3–11.