Monitoring Cyber-Physical Systems using a Tiny Twin to Prevent Cyber-Attacks

Fereidoun Moradi*[1], Maryam Bagheri[2], Hanieh Rahmati[3], Hamed Yazdi[4], Sara Abbaspour Asadollah[1], and Marjan Sirjani[1]

[1] School of Innovation, Design and Engineering, Mälardalen University, Västerås, Sweden
[2] Tehran Institute for Advanced Studies, Khatam University, Tehran, Iran
[3] University of Tehran, Tehran, Iran
[4] Chavoosh ICT, Isfahan, Iran
{fereidoun.moradi, sara.abbaspour, marjan.sirjani}@mdu.se
mbagheri@ce.sharif.edu, rahmati_hanie@ut.ac.ir, h.yazdi@chavoosh.com

Abstract. We propose a method to detect attacks on sensors and controllers in cyber-physical systems. We develop a monitor that uses an abstract digital twin, Tiny Twin, to detect false sensor data and faulty control commands. The Tiny Twin is a state transition system that represents the observable behavior of the system from the monitor point of view. At runtime, the monitor observes the sensor data and the control commands, and checks whether the observed data and commands are consistent with the state transitions in the Tiny Twin. The monitor produces an alarm when an inconsistency is detected. We model the components of the system and the physical processes in the Rebeca modeling language and use its model checker to generate the state space. The Tiny Twin is built automatically by reducing the state space, keeping the observable behavior of the system, and preserving the trace equivalence. We demonstrate the method and evaluate it in detecting attacks using a temperature control system.

Keywords: Monitoring · Model Checking · Abstraction · Cyber-Physical Systems · Attack Detection and Prevention · Cyber-Security.

1 Introduction

Cyber-Physical Systems (CPSs) are mostly safety-critical systems that integrate physical processes in the industrial plants (e.g., thermal power plants or smart water treatment plants) with sensors, actuators and controller components. Since these components are integrated via a communication network (usually wireless), a CPS is vulnerable to malicious cyber-attacks that may cause catastrophic damage to the physical infrastructure and processes. Cyber-attacks may be performed over a significant number of attack points and in a coordinated way. So, detecting and preventing attacks in CPSs are of significant importance.

Intrusion Detection Systems (IDSs) are deployed in communication networks to defend the system against cyber-attacks. Regular IDSs cannot easily catch

complex attacks. They need to be equipped with complicated logic, based on human (safety and security engineers) reasoning [26]. In rule-based IDSs [26], a set of properties that are extracted from the system design specification are considered as a rule-set to detect attacks. Indeed, if an IDS finds a deviation between the observed packets in the network and the defined rules, it produces an alarm and takes a predefined action such as dropping the packets. The key challenge is the effort required to specify the correct system behavior as rules.

In this paper, we propose a method to detect cyber-attacks on sensors and controllers of a CPS. We model the system components, the progress of time and the interactions between components, and then abstract the model based on the observable behavior of the system. The observable behavior is the set of events that can be observed (sensor data) and controlled (control commands) by the controllers. We develop a monitor module as an IDS that employs the created abstract model to detect the cyber-attacks. The monitor walks over the abstract model at runtime to check whether the behavior of the system is consistent with the model generated at design time. Our approach is similar to using MAPE-K architecture and models@runtime that support the monitoring and adaptation at runtime [6]. Digital Twins [10] are used as digital representation of the system to advance the system monitoring. We call our abstract model a Tiny Twin as it resembles an abstract version of a Digital Twin, and we use it as models@runtime in our method.

To build the Tiny Twin, we start with developing a Timed Rebeca model [32] of the system, and create the state space using a model checker tool. The state space is often huge and usually contains state transitions reflecting the events that are not observable for the controllers, therefore it cannot be directly used in the monitoring. We then reduce the state space using our abstraction tool while preserving the trace equivalence between the original transition system and its abstracted version. In the abstraction process, we consider the observable actions to be receiving sensor data by the controllers, and sending commands to the actuators. These actions are related to the labels of the corresponding transitions in the state space. Using our abstraction tool, the state variables in the controller that store the receiving sensor data as well as the state variables in the actuators that reflect the changes on the state of the actuators will stay observable.

There is a rich literature on using formal models to detect and prevent cyber-attacks on CPSs (e.g., [21,5,7,14,34,11]). For instance, authors in [21] define the behavior of the system using an automaton and employ it to detect man-in-the-middle attacks. Authors in [5] build automaton models, determine the set of unsafe system states, and check whether the system under attacks reaches these states. Authors in [19] model the system as finite state machines and verify the system behavior at runtime. In our work, we integrate the system monitoring with a state space model to detect attacks. The advantage of our method is that the model used by the monitor (the Tiny Twin) is automatically derived from the Timed Rebeca model [32], not purely a specification, and then reduced based on the observable state transitions. To simulate and check the attack detection

process, the Timed Rebeca model is mapped to an executable code in Lingua Franca [23]. The mapping does not require much effort because both languages use the same actor-based semantics. Building an actor-based model based on the specification and requirements, then model check and debug it, can reveal the inconsistencies and ambiguities in the specification. Going all the way to the executable code and apply the necessary revisions, helps in reflecting back the necessary details in the model.

In different situations, we may decide differently, for example keeping the signals passed between different controllers as the observable events in the abstract state transition system of a distributed system. We also include logical time in the Tiny Twin model which is related to the physical time with a limited amount of deviation (guaranteed by Lingua Franca framework). Some of the meta-rules regarding the system can be captured in the Tiny Twin. For example, the fact that "seeing extreme changes in the temperature in a short period of time is not possible" can be captured in the way we model the environment, and then reflected in the state space and its abstracted version (Tiny Twin).

Contribution. We develop an abstraction tool that 1) reduces the state transition system of a CPS (i.e., the state space which is the output of a model checker tool) based on the observable behavior (a list provided by the modeler). 2) We propose and implement a monitor algorithm that uses the Tiny Twin to detect cyber-attacks. 3) We develop a temperature control system case study in Lingua Franca [23]. 4) We evaluate the method by simulating several attacks and show how the monitor can catch them, and how the Tiny Twin model helps.

Outline. We give an overview of our method in Section 2. We introduce Timed Rebeca and Lingua Franca in Section 3. We present our abstraction tool and explain our monitor algorithm in Section 4. In Section 5, we describe a temperature control system and evaluate our method by implementing the monitor module in Lingua Franca. Section 6 covers the related works. The conclusion and future directions of this work are discussed in Section 7.

2 Overview of our Approach

The overview of our approach is shown in Fig. 1. In [28], we explained our method to develop the model of a CPS in Timed Rebeca [32] and use the Afra model checker [1] to verify the model. In [28], it is shown that how entities of a CPS, i.e., sensors, actuators, controllers, and physical plant are modeled as actors, and interactions between them are modeled as messages passed between the actors. In this paper, we develop an abstraction tool (part (A) in Fig. 1) that reduces the state space of the Timed Rebeca model (generated by Afra model checker tool) to create the Tiny Twin based on the observable behavior for the controllers in the system. The state space of a Timed Rebeca model is a state transition system [33], in which a state represents a particular configuration of the system and includes values of state variables of the actors, and a transition represents triggering of an event (handling a message received by an actor) or progressing the logical time of the model. We develop a monitor module that uses

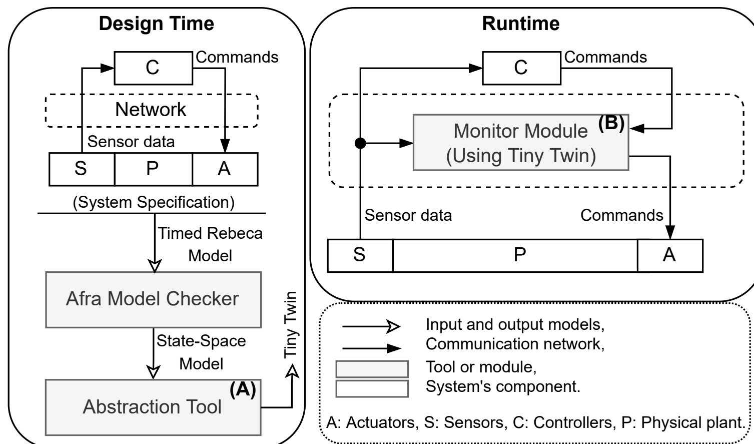

Fig. 1: The overview of our approach. The state space of the Timed Rebeca model of a CPS is generated by the Afra model checker and is reduced by our abstraction tool (see Sec. 4.1). The result is a Tiny Twin that is used by our monitor (see Sec. 4.2) to detect the attacks. The monitor executes together with the system at runtime.

the created Tiny Twin to track the order and the timing of events (part (B) in Fig. 1). In safety-critical systems, an isolated and trusted hardware component is leveraged to enhance the security of the complete system [27]. We implement our monitor algorithm as a module that can run on an isolated platform with hardware security support [2].

We implement our monitor module in Lingua Franca [23] that is a language for programming CPSs. In principle, the Lingua Franca code can connect to the physical plant and the controller through the input/output communication channels in the actual system. In this paper, we use Lingua Franca to simulate the system at runtime and evaluate the detection capability of our method by defining compromised components. As shown in Fig. 1, the monitor module observes the sensor data entering the controller, and the control commands leaving to the actuators. We rely on the logical time provided in Lingua Franca when we compare the system execution time with the time represented in the Tiny Twin. Lingua Franca relates logical time to physical time based on the approach introduced in Ptides/Spanner design [8,35]. In the Lingua Franca program we have a network of actors and a scheduler handles the order of events. The scheduler watches the local clock of each actor and hold off processing the message until its measurement of physical time exceeds a threshold. This threshold is defined to align two timelines, logical time and physical time [22].

The Tiny Twin defines the observable behavior of the system in the absence of an attack and contains the order and the time at which the sensor data and control commands are communicated. Transitions in Tiny Twin are tagged by a label that indicates an action or the progress of time. To detect an attack, the monitor receives the sensors data and the control commands at runtime,

compares the sensors data with the values of the state variables and checks whether the control commands are consistent with the outgoing transitions in the Tiny Twin. If this is the case, the monitor compares the time of the current state of the Tiny Twin with the time at which the sensor data or the control commands are received. If the monitor observes an unexpected sensor data or control command at an unexpected time, it raises an alarm and drops the faulty control command.

3 Background: Timed Rebeca and Lingua Franca

In this section, we provide an overview on Timed Rebeca and describe the Lingua Franca programming language.

Timed Rebeca. Rebeca [31] is an actor-based modeling language for modeling and formal verification of concurrent and distributed systems. Actors, called *rebecs*, are instances of *reactive classes* and 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 that is a global time synchronized between all actors. Each actor has a set of variables. Besides, it has 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 method that is called *message server*. The actor can change 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 method. The generated state space of the Timed Rebeca model contains two or more outgoing transitions at the same time for each state in which the value of variables are nondeterministically changed.

Timed Rebeca is supported by the Afra model checker tool [1]. Afra generates the state transition system (i.e., state space model) of the model where states contain values of variables of actors along with the logical time, and transitions represent progressing the logical time of the model or taking a message from an actor's bag and executing the corresponding method [30,16]. The transition in the latter case is labeled with the name of the executed message server. The transition in the first case is labeled with the amount of time progress and is called a timed transition. Note that a model has an unbounded state space when the logical time goes to infinite. In [16], the authors propose an approach, called Shift Equivalence Relation, to make the state space of a Timed Rebeca model bounded, where possible. Afra implements this approach to generate the finite state space of the model.

Lingua Franca. Lingua Franca is a meta language based on the Reactor model for programming CPSs [24,22]. A Reactor model is a collection of *reactors*. A reactor has one or more routines that are called *reactions*. Reactions define the functionality of the reactor, and have access to a *state* shared with other reactions, but only within the same reactor. Reactors have named (and typed) *ports* that allow them to be connected 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 [12], except rather than responding to messages, reactions are triggered by discrete events and may also produce them. An event relates a value to a *tag* that represents the logical time at which the value is present. 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 Lingua Franca, 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 timer triggers at the start time of the execution), and then triggers every *period*. Lingua Franca has a built-in type for specifying time intervals. A time interval consists of an integer value accompanied with 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. Each Lingua Franca program contains a *main reactor* that is an entry point for the execution of the code.

4 Abstraction Tool and Monitor Algorithm

In this section, we first present our developed abstraction tool, which reduces the state space of a Timed Rebeca model based on the observable behavior for the controllers in the system and creates the Tiny Twin. Then, we explain how our monitor detects attacks by checking the consistency between the observed sensor data and control commands and the state transitions in the Tiny Twin.

4.1 Abstraction Tool

Our abstraction method is implemented in a tool [1] by considering the reduction algorithm proposed by Jansen et al. [13]. In order to create an abstract model of a Timed Rebeca model, the modeler provides the tool a list of variables whose values are changed by the observable actions. The tool reduces the state space of a Timed Rebeca model based on the given list. It is applied to the original state space, preserves observable traces (i.e., sequences of actions that represent the observable behavior) where it iteratively refines indistinguishable states, i.e., the classes containing equivalent states, while hides transitions that are called silent transitions. The abstract and original models of the system show the same observable behavior when hiding silent transitions. It begins at the initial state and traverses the outgoing transitions one by one (i.e., BFS

[1] https://github.com/fereidoun-moradi/Abstraction-tool

graph search). It merges a pair states into an equivalence class if they have the same values for the given variables (the time variable in each state is preserved in the abstraction process). All transitions that modify the given variables in the list stay observable and other transitions that do not change the values of these variables become silent transitions.

Example. Fig. 2 illustrates how the abstraction tool performs on an example. We depict the transition system of a Timed Rebeca model with the set of variables $\{s, w, h\}$ in Fig. 2(a). We show the values of variables in each state and use $time+ = 10$ to denote that the logical time progresses by 10 units of time over a transition. The transitions that are not timed transitions are labeled with a label of $\{getsense$, $activate_h$, $heating$, $switchoff\}$. The notation $(a \gg b)$ on each transitions denotes that the source state has the time value a (the value of variable now), which is shifted by the value b and becomes the time value at the leading state. In this system, we may want to check properties such as "the command $activate_h$ will be issued if $s = 20$ and $h = false$" and "the command $switchoff$ will be issued in less than 10 units of time if $s = 20$ and $h = true$". Two paths $\{activate_h\}$ and $\{heating, getsense^*, switchoff\}$, respectively, from state $S3$ to state $S4$ and from state $S4$ to state $S2$, satisfy these properties. The action *heating* is an unobservable action. The tool receives $V = \{s, h\}$ to reduce the transition system of Fig. 2(a). These state variables reflect the changes occur in the system after the observable actions $\{getsense, switchoff\}$ are executed. The equivalence classes created by the tool and the resulting abstract model is shown in Fig. 2(b). Our abstraction tool names each state of Fig. 2(b) with GSi that corresponds to the class e_class_i. For instance, e_class_6 is replaced with $GS6$. Two paths are preserved in the abstract transition system, respectively, from class $GS2$ to class $GS3$ and from $GS3$ to class $GS1$.

4.2 Monitor Algorithm

Algorithm 1 shows the pseudo-code of our *monitor* algorithm. The algorithm observes the sensor data and the control commands transmitted in the network and decides to drop or pass the control commands to the actuators. Suppose that p number of sensors send the sensed data $y = \{y_1, ..., y_p\}$ to the controller. The time k indicates a time value which is derived using Lingua Franca code and it is advanced based on the logical timeline defined in the language. The commands $u = \{u_1, ..., u_n\}$ are issued by the controller for n actuators to maintain the physical plant in the certain desirable state. The sensor data are all within a defined range, i.e., $\forall i \in [1, p], y_i \in [y_i^{min}, y_i^{max}]$.

The algorithm gets (S, T) as an input, where S and T are respectively the sets of states and transitions in the Tiny Twin. The algorithm returns commands u or produces an $alarm$. The monitor starts its observation when the system executes. Upon receiving sensor data y and/or commands u, the algorithm compares y and v (the values stored in the state variables of the model and correspond to the sensors, i.e., $v = \{v_1, v_2..., v_p\}$) at the current state s in the Tiny Twin. If the algorithm finds no differences between them, it proceeds and checks whether the

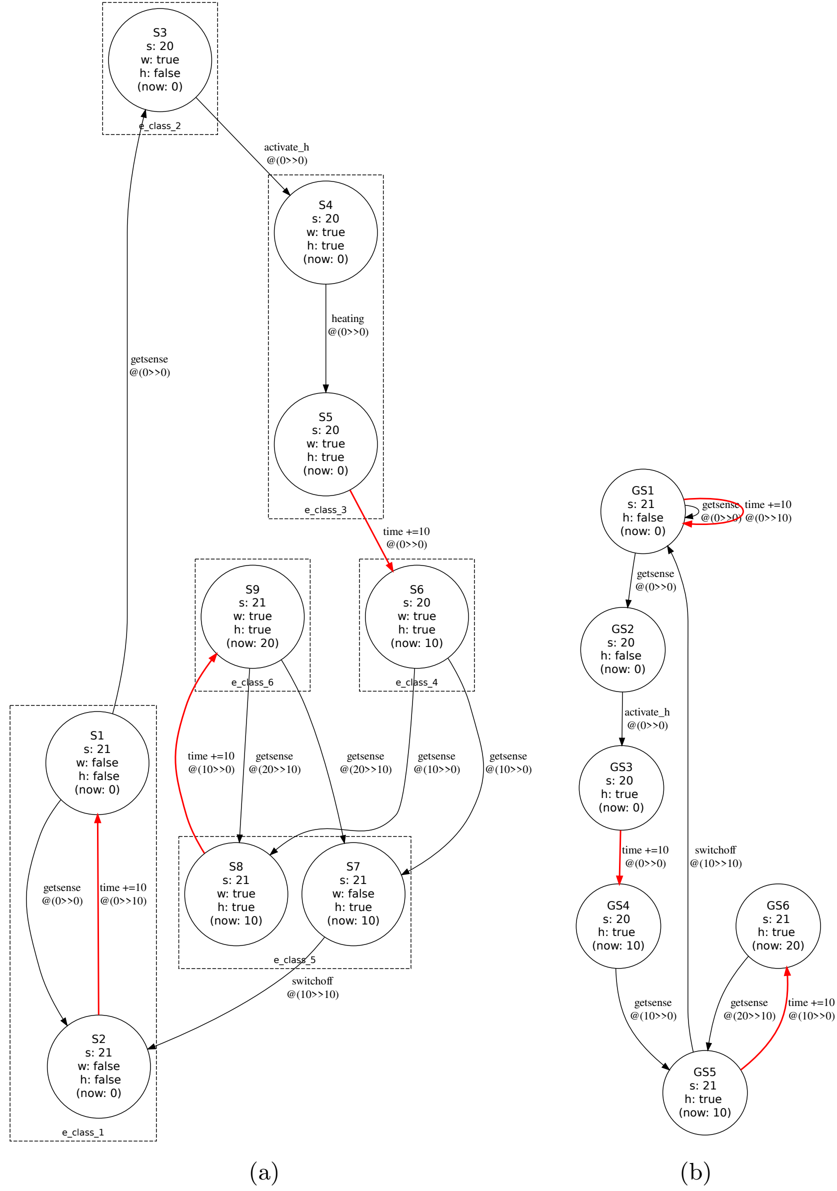

Fig. 2: (a) The transition system of an example Timed Rebeca model with the equivalence classes created by the abstraction tool. (b) The Tiny Twin of the transition system is depicted in Fig. 2(a).

commands u are consistent with the corresponding transitions in the model. If this is the case, the algorithm sends the commands u to the actuators. Otherwise, the algorithm produces an alarm and terminates the process of monitoring.

In the following, we explain the details of the algorithm. The algorithm sets the current state of the system to the initial state of the model that is $s_0 \in S$ (line 2). The algorithm begins its iteration by observing the sensor data y and the commands u (line 3). We use the function *getTime* to obtain the logical time of the current state and put it in the time variable x (line 4). The algorithm checks the branching states if it observes the sensor data y (lines 6-7). The function *traverse* returns the next state by traversing the given transitions (line 8). The function *cmpSensorData* compares the data y and v and the function *cmpTimes* compares the time k (at which y are observed) with the time x in the model (line 9). If y and v are the same and are observed at time x (the order in which the sensor data is observed is consistent with the order of transitions in the model), the algorithm proceeds and repeats the monitoring process (lines 10). Otherwise, the algorithm returns an alarm (line 11) and terminates the process of monitoring. The algorithm then compares the commands u with the corresponding transitions in the model at time x (line 12). This comparison is performed by the functions *checkTransitions* and *cmpTimes* (line 13). The function *checkTransitions* extracts labels of the outgoing transitions of the current state and checks whether the commands are equal to these labels. If this is the case, the algorithm uses the function *traverse* to reach the next state and returns the commands u (lines 14-15). Otherwise, it drops the commands and produces an *alarm* (lines 17-18). The current state is updated if the time k is advanced (lines 19). The algorithm repeats its monitoring.

Example. Let the Tiny Twin of Fig. 2(b) be the input model of Algorithm 1. The algorithm sets the current state to $GS1$. It observes the sensed value 20 and the control command *activate_h* at time k, i.e. $k = 0$. The monitor traverses the transition *getSense* and sets $GS2$ as the current state. The monitor compares the sensed value and the value of the state variable in the current state. If the values are the same and the logical time of the current state is equal to k, i.e. $x = 0$, the monitor proceeds and checks the label of the outgoing transition. It compares the command and the *activate_h* label. Since the command and the label are the same, the monitor traverses the outgoing transition and sets the current state to $GS3$. The current state has an outgoing timed transition, the monitor repeats its monitoring and waits to observe the logical time k advances. The monitor observes a new sensed value 21 and control command *switchoff* at time $k = 10$. The monitor traverses the timed transition and sets the current state to $GS4$. It traverses *getSense* transition and compares the sensed value and the value of the state variable. It checks whether the values are the same and the logical time of the current state is equal to k, i.e. $x = 10$. The monitor proceeds and checks the label of the outgoing transition, i.e., *switchoff* label. Since the command and the label are the same, the monitor traverses the outgoing transition *switchoff* and sets the current state to $GS1$. The monitor repeats the monitoring process. In $GS5$, if the monitor observes a new sensed value 21 at

Algorithm 1: Monitor algorithm

Input: An abstract state space (S, T)

Output: Commands u, or an *alarm*

```
1  begin
2    s ← s_0 ∈ S;
3    while observes y, u or time k do
4      x ← getTime(s);
5      if y is present then
6        s' ← s;
7        for each leading state of s' do
8          s ← traverse(y, s', S, T);
9          if cmpSensorData(y, v) and cmpTimes(k, x) then
10           break;
11       return alarm;
12     if u is present then
13       if checkTransitions(u, s, S, T) and cmpTimes(k, x) then
14         s ← traverse(u, s, S, T);
15         return u;
16       else
17         drops(u);
18         return alarm;
19     s ← timeProgress(s, S, T);
```

time $k = 20$, without observing a control command, it traverses the timed transition and sets the current state to $GS6$. It then traverses *getSense* transition and sets the current state to $GS5$. The monitor updates the logical time with the time value in $GS5$ and the amount of time progress, i.e., $k = 20$ (i.e., shift equivalence relation between states $GS6$ and $GS5$). It compares the sensed value and the value of the state variable in $GS5$. The monitor repeats the same process by observing the sensor data or control commands. The monitor produces an alarm and terminates the monitoring process if it observes a sensed value or a control command inconsistent with the model. It returns an alarm containing a tuple $[k, y^i, u^d, v_1, v_2, ..., v_w]$ where k is a time value showing at which time during system execution an inconsistency is identified, y^i is the inconsistent sensor data, u^d is the dropped control command and v_i are values of state variables in the state GSi where the *monitor* terminated the system execution.

5 Case Study: a Temperature Control System

We evaluate the applicability of our method in detecting and preventing cyber-attacks using a temperature control system. The goal of attacks is to change the temperature out of the desired range or cause damage on the physical infrastructure (i.e., the heating and cooling unit). We assume that attackers can send false sensor data or compromise the controller to alter the commands issued by the controller. We developed the Timed Rebeca model of the temperature control

system (see Listing 1(a)) in which four reactive classes are defined to specify the system components and the physical process. We use the Afra model checker to produce the state space of the developed Timed Rebeca model and exploit our abstraction tool to generate the Tiny Twin. We implement both the system and the monitor module in Lingua Franca. We use the mapping between Timed Rebeca and Lingua Franca [33] to write a Lingua Franca code of the system (see Listing 1(b)). This code can be executed on a single core, on multiple cores, or on separate processors connected via a network. In this case study, we configure the number of threads to 1 since the code is mapped form a Rebeca model in which each actor has a single thread of execution. The complete codes of the system and the monitor are available on GitHub [1].

The temperature control system is responsible for maintaining the temperature of a room at a desired range (i.e., the values between 21 and 23). This system includes a sensor, a hc_unit (heating and cooling unit) as an actuator, and a controller (lines 46,56, and 7 in Listing 1(a)). The controller receives sensor data from the sensor and transmits the activate_c, activate_h or switch off command to the hc_unit to respectively activate the cooling or heating process, or switch the heating/cooling process off (lines 10-23). Assume that there is a window inside the room (line 30) and the outside air blows inside when the window is open (line 33). The controller does not know whether the window is open or closed but can activate the heating/cooling process based on the sensed temperature value. The cooling process is activated if the temperature value is higher than the desired range (e.g., the value 24) (line 21). The heating process is activated if the temperature value is lower than the desired range (e.g., the value 20) (line 17). The heating/cooling process is switched off if the temperature value is regulated to the desired range. The physical process is temperature regulation (lines 37-44), and the desired state is a specific range for the temperature. We assume that the temperature of the room is within the desired range at the beginning (i.e., the value 22) (line 1).

Similar to the Timed Rebeca model of the system, the Lingua Franca code implements all components of the system (Listing 1(b)). The input port *getSense* in the reactor *controller* (line 3) is defined to get a sensor value, and three output ports *activate_h*, *activate_c*, and *switchoff* (lines 4-6) are defined to send values as commands to the *hc_unit*. We set the value of *activate_h* to 1 to trigger the heating (line 18), and the value of *switchoff* to 0 to trigger the switch off in the *hc_unit* (line 14). The main reactor instantiates the components and binds their input and output ports to connect the components together (line 67). For example, we connect the output port *out* in the reactor *sensor* to the input port *getSense* in the reactor *controller* (line 70). This way, the new temperature value is transferred from the sensor to the controller. In the main reactor, the use of *after* indicates that a value reaches the input port of the reactor *controller* after 10 units of time (line 70). Note that we use a time function to measure the logical time elapsed since the code started to run (line 37).

[1] https://github.com/fereidoun-moradi/RoomTemp

```
1   env int desiredValue = 22; //environment variables
2   env int timingInterval = 10;
3   reactiveclass Controller(8){
4     knownrebecs{ HC_Unit hc_unit; Sensor sensor;}
5     statevars{ int sensedValue;
6       boolean heating; boolean cooling;}
7     Controller(){
8       sensedValue = desiredValue;
9       heating = false; cooling = false;}
10    msgsrv getSense(int temp){
11      sensedValue = temp;
12      if (temp <= 23 && temp >= 21) { //desired range
13        if (heating == true || cooling == true) {
14          hc_unit.switchoff();
15          heating = false; cooling = false;
16        } else { sensor.start();}
17      } else if (temp < 21) {
18        if (heating == false) { //control command
19          hc_unit.activate_h(); heating = true;
20        } else  { sensor.sense(sensedValue); }
21      } else if (temp > 23) {
22      //...
23      }
24  }
25  reactiveclass Room(8){
26    knownrebecs{ Sensor sensor;}
27    statevars{ int temperature; int outside_air_blowing; }
28    Room(){
29      temperature = 22; //initial value
30      outside_air_blowing = 0; //window is closed
31    }
32    msgsrv status() { //nondeterministic assignment
33      outside_air_blowing = ?(1,0,-1);
34      temperature = temperature - outside_air_blowing;
35      sensor.sense(temperature);
36    }
37    msgsrv heating() {
38      temperature = temperature + 1;
39      self.status();
40    }
41    msgsrv cooling() {
42      temperature = temperature - 1;
43          self.status();
44    }
45  }
46  reactiveclass Sensor(8){
47    knownrebecs{ Room room; Controller controller;}
48    Sensor(){ self.start();}
49    msgsrv start(){
50      room.status();
51    }
52    msgsrv sense(int temp) { //sensing intervals
53      controller.getSense(temp) after(timingInterval);
54    }
55  }
56  reactiveclass HC_Unit(8){
57    knownrebecs{ Room room;}
58    statevars{
59      boolean heater_on;
60      boolean cooler_on;}
61    HC_Unit() { heater_on = false; cooler_on = false;}
62    msgsrv activate_h() {
63      heater_on = true; cooler_on = false;
64      room.heating(); //heating
65    }
66    msgsrv activate_c(){
67      cooler_on = true; heater_on = false;
68      room.cooling(); //cooling
69    }
70    //...
71  }
72  main{
73    Room room(sensor):();
74    Controller controller(hc_unit,sensor):();
75    Sensor sensor(room,controller):();
76    HC_Unit hc_unit(room):();
77  }
```

(a) Timed Rebeca model

```
1   target Cpp {fast: false, threads: 1};
2   reactor Controller { //input and output ports
3       input getSense:int;
4       output activate_h:int;
5       output activate_c:int;
6       output switchoff:int;
7       state heating:bool(false);
8       state cooling:bool(false);
9       reaction(getSense) ->
10       activate_c, activate_h, switchoff {=
11       if(*getSense.get() <= 23 &&
12                  *getSense.get() >= 21){
13            if(heating == true || cooling == true) {
14              switchoff.set(0);
15              heating = false; cooling = false;}
16       } else if(*getSense.get() < 21){
17            if(heating == false){
18              activate_h.set(1); heating = true;
19            } else { activate_h.set(0); }
20       } else if(*getSense.get() > 23) {
21       //...
22       }
23      =}
24  }
25  reactor Room {
26      input cooling:int;
27      input heating:int;
28      input status:int;
29      output sensedValue:int;
30      state temperature:int(22);
31      state outside_air_blowing:int(0);
32      reaction(status) -> sensedValue {=
33        outside_air_blowing = rand() % 3 + (-1);
34        temperature =
35            temperature - outside_air_blowing;
36        sensedValue.set(temperature);
37        auto elapsed_time =
38           get_elapsed_logical_time();
39      =}
40      reaction(heating) {=
41       temperature = temperature + *heating.get(); =}
42      reaction(cooling) {= //...
43      =}
44  }
45  reactor Sensor {
46      input sensedValue:int;
47      output out:int;
48      output sense:int;
49      timer start(0, 1 sec);
50      reaction(start) -> sensedValue {=
51          sense.set(1); =}
52      reaction(sensedValue) -> out {=
53          out.set(sensedValue.get());=}
54  }
55  reactor HC_Unit {
56      input activate_h:int;
57      input activate_c:int;
58      input switchoff:int;
59      output heating:int;
60      output cooling:int;
61      reaction(activate_h) -> heating {=
62          if (*activate_h.get() == 0){
63              heating.set(0);
64          } else { heating.set(1); }
65      =} //...
66  }
67  main reactor RoomTemp {
68    //...
69    room.sensedValue -> sensor.sensedValue;
70    sensor.out -> controller.getSense  after 10 sec;
71    sensor.sense -> room.status;
72    unit.heating -> room.heating;
73    unit.cooling -> room.cooling;
74    controller.activate_h -> unit.activate_h;
75    controller.activate_c -> unit.activate_c;
76    controller.switchoff -> unit.switchoff;
77  }
```

(b) Lingua Franca code

Listing 1: Timed Rebeca model (a) and Lingua Franca code (b) of the temperature control system.

5.1 Tiny Twin

Fig. 3 shows the Tiny Twin of the state space of the developed Timed Rebeca model for the temperature control system. The Tiny Twin is generated by the abstraction tool based on the list V={*sensedValue*, *cooler_on*, *heater_on*} of state variables. The original state space of the model includes 76 states and 103 transitions while the generated Tiny Twin contains 21 states (i.e., equivalence classes) and 36 transitions. The Tiny Twin is trace equivalent to the original state space (projected on the variables containing sensors data and control commands). The values of state variables in the list V are shown on each state of the model. The variables are indicated by the first letter of the variable names in Fig. 3. The variable *now* shows the logical time of the model. The transition between two states is either labeled with an action or the progress of time (i.e., *time+=10*). In the initial state (GS1), the stored temperature value in the controller is within the desired value, i.e., *s: 22*, and the cooling and heating processes have been switched off, i.e., *c: false* and *h: false* (GS1). The logical time of the model progresses by 10 units (GS2). The controller receives a new sensor data that its value depends on the current temperate in the room (GS3 and GS4). The controller begins a new cycle of temperature regulation by reading sensor data that indicates the current temperature value is higher/lower than the desired range (GS8 and GS9). The controller sends *activate_c*/*activate_h* command to the hc_unit for activating the cooling/heating process, i.e., *c: true* or *h: true* (GS11 and GS12). The temperature value is regulated by the heating/cooling process and reaches the desired range if the window is closed (GS15 and GS17). Otherwise, the temperature value is regulated further by keeping the heating/cooling process activated. In addition to the activation of the heating/-cooling process, the outside air blowing inside through the open window causes the temperature to increase/decrease (GS13 and GS14). The controller sends the *switchoff* command if the new sensor data is within the desired value and the heating/cooling process has been activated (GS16 and GS18). If the sensed temperature value is higher/lower than the desired range, the controller does not send any command to the hc_unit because the cooling/heating process has been activated (GS13 and GS14).

5.2 Attack Types and Detection Capability

We evaluate the capability of the developed *monitor* module in detecting attacks. We consider three types of attacks that target the *integrity* aspect of a CPS. (1) Attackers have the ability of tampering sensor data or injecting any arbitrary values into the vulnerable channel between controller and sensors, i.e., *replay or tampering attack*, (2) attackers are able to manipulate the controller through malicious code injection into the software of the controller, i.e., *fabrication or masquerade attack*, and (3) one or more attackers can perform a coordinated attack to force the system to change its correct functionally. Any of the above attacks could be performed in a stealthy way when attackers try to remain

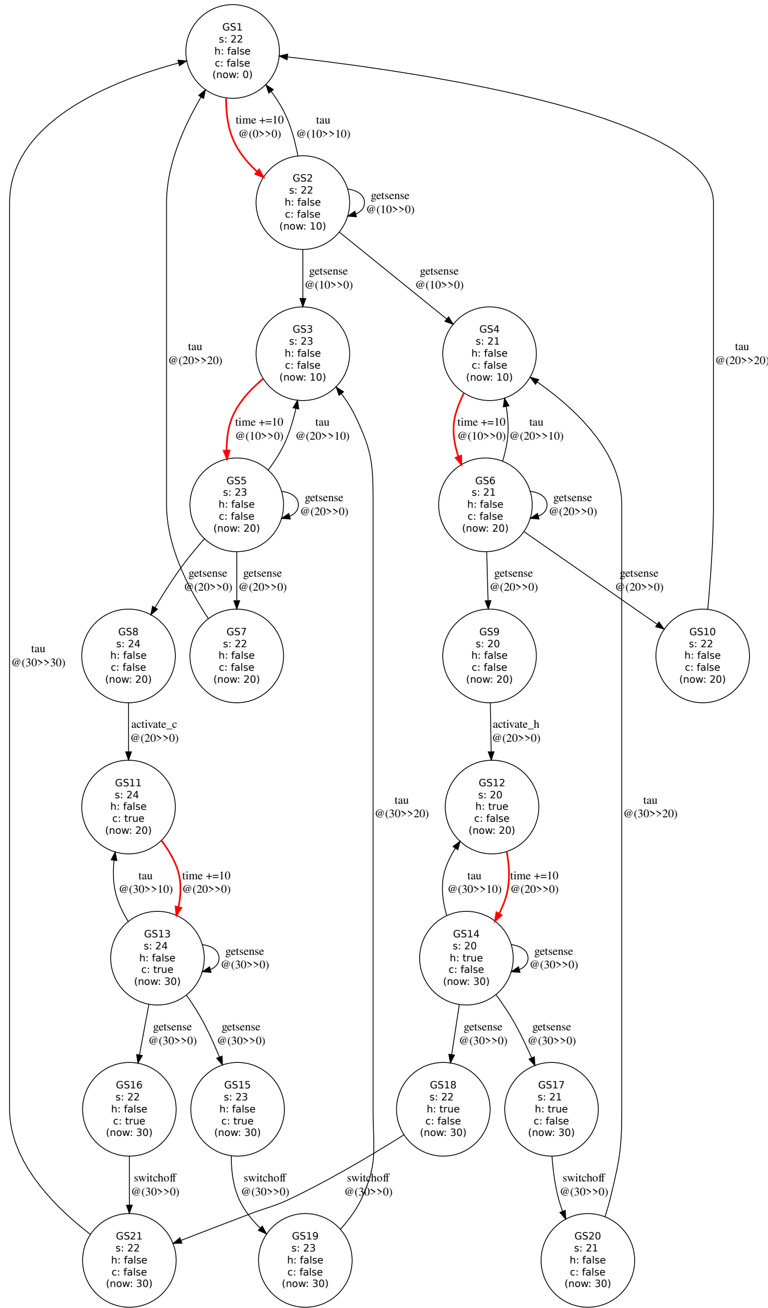

Fig. 3: The Tiny Twin of the temperature control system. The labels on each state show the temperature value s and the status of the cooling c and heating h processes. The variable *now* shows the logical time on each state. The transition between two states is labeled with an action or the progress of time.

undetected by doing slow damage and keeping the impact of the attack close to the changes in the correct behavior of the system.

According to [28], we model these types of attacks using the defined attack schemes. To implement the attacks, we modify the reactions of the *sensor* and the *controller* reactors in the developed Lingua Franca code. This way, these reactors behave as *compromised components* and respectively send false sensor data and faulty control commands on the output ports.

We consider the number of false sensor data and faulty control commands as the number of attacks. In our experiments, we simulate 20 false sensor data and 12 faulty control commands as listed in Table 1. We also simulate 240 coordinated attacks (combination of the false sensor data and the faulty control commands). For each attack, we execute the Lingua Franca compiler once and generate an executable file. We calculate the detection rate of the monitor with respect to the detected/undetected attacks. In this case study, the detection rate is around 68.8 percent and the average time of the state checking of the monitor is around 0.0008 seconds.

Table 1: Attacks and detection capability of the monitor module.

System States	*# Attacks*	*False sensor data/ Faulty control commands*	*Detection Capability* (DS/DC)
GS1 and GS2	4	Sensor data (20, 21, 23, or 24)	DS (20 and 24)
GS3 and GS5	4	Sensor data (20, 21, 22, or 24)	DS (20 and 21)
GS4 and GS6	4	Sensor data (20, 22, 23, or 24)	DS (23 and 24)
GS8	2	Command (*activate_h* or *switchoff*)	DC (*activate_h* and *switchoff*)
GS9	2	Command (*activate_c* or *switchoff*)	DC (*activate_c* and *switchoff*)
GS11 and GS13	4	Sensor data (20, 21, 22, or 23)	DS (20 and 21)
GS12 and GS14	4	Sensor data (21, 22, 23, or 24)	DS (23 and 24)
GS15, GS16, GS17, GS18	2	Command (*activate_h* or *activate_c*)	DC (*activate_h* and *activate_c*)

#Attacks.: Number of simulated attacks, *DS*: Detect false sensor data, *DC*: Detect faulty control commands.

Table 1 shows the states with one or more outgoing transitions that correspond to the sensor data or control commands. If the *compromised controller* sends a command that differs from the outgoing transition, the *monitor* module can detect/drop the faulty control command. From states $GS2$, $GS5$, $GS6$, $GS13$ and $GS14$ (see Fig 3) you may move to different states. For instance, assume that 23 is sensed as the temperature value in $GS2$ but the *compromised sensor* sends the value 20. According to the Tiny Twin of the case study, the value for the next states can be either 21 ($GS4$), 22 ($GS1$ or $GS2$), or 23 ($GS3$) so the *monitor* module detects the false sensor data. Note that the controller should in principle sends *activate_h* to activate the heating process by sensing 20. 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 22 to 20. So, this is captured as an unexpected behavior. As another example, assume that the value 22 is sensed as the temperature value in $GS2$ but the *compromised sensor* sends a sensed value 23 or 21. In this case, the *monitor* module can not detect the false sensor data. We are able to use meta-rules to check if the paths between turning the heating (or cooling) unit(s) are taken too quickly, or any of these units stay turned on for a time longer than expected.

Table 2: Alarms of the monitor module in case of attacks.

System States	*False sensor data/ Faulty control commands*	*Alarms list*
GS1 and GS2	Sensor data (20)	$[time, y^i : 20, y : 23, s : 22, c : false, h : false]$
GS3 and GS5	Sensor data (21)	$[time, y^i : 21, y : 22, s : 23, c : false, h : false]$
GS4 and GS6	Sensor data (23)	$[time, y^i : 23, y : 22, s : 23, c : false, h : false]$
GS8	Command (*activate_h*)	$[time, u^d : activate_h, y : 24, s : 24, c : false, h : false]$
GS9	Command (*switchoff*)	$[time, u^d : switchoff, y : 20, s : 20, c : false, h : false]$
GS11 and GS13	Sensor data (21)	$[time, y^i : 21, y : 22, s : 24, c : true, h : false]$
GS12 and GS14	Sensor data (24)	$[time, y^i : 24, y : 22, s : 20, c : false, h : true]$
GS16	Command (*activate_c*)	$[time, u^d : activate_c, y : 22, s : 22, c : true, h : false]$

time: The logical time which is derived using Lingua Franca code.

Table 2 shows the alarms list returned by the monitor module when a false sensor data or a faulty control command is detected. The alarm is comprised of a time value, a false sensor data or a faulty control command, the status of the physical plant reported by the sensor and the value of the state variables in the state where the monitor module terminated the system execution. Having this report would be very helpful for system testers/developers to find the situation of the system state when the alarm happened and find the actual source of the attack.

In a CPS, there may be several variables involved in the physical process as well as various sensors and actuators. The monitoring approach using the Tiny Twin enables us to consider only variables are affected during an attack (i.e., violation of properties). Tiny Twin provides relevant information about attacks that can be employed in mitigation techniques, backtracking and recovering the system after attacks. We have developed the Timed Rebeca models and the Lingua Franca codes of two case studies (Secure Water Treatment system (SWaT) [1] and Pneumatic Control System (PCS) [2]) available on the GitHub, for which the *monitor* module can properly detect attacks on the system. In these case studies, the original state space model of the Timed Rebeca model of the SWaT contains 614 states and 777 transitions and the original state space model of the PCS has 1388 states and 2686 transitions. The Tiny Twin models of these systems respectively have 85 states and 139 transitions and 120 states and 224 transitions.

6 Related Work

There are interesting works based on formal methods and also using models for detecting attacks at runtime.

Lanotte et al. [18] propose a formal approach based on runtime enforcement to ensure specification compliance in networks of controllers. They define a synthesis algorithm with respect to Ligatti et al.'s edit automata [20]. The algorithm takes an alphabet of observable actions and a timed correctness property, and

[1] https://github.com/fereidoun-moradi/SWaT-Rebeca-Model
[2] https://github.com/fereidoun-moradi/Reconfigurable-Pneumatic-System

returns an edit automaton as an enforcer. In their work, the enforcers are synthesized regarding deterministic behaviors of the controllers. The network of enforcers preserves weak bisimilarity equivalence in relation to the networks of controllers. The enforcers contain clock variables and specify safe behaviors of controllers. At runtime, they are used to detect the compromised controllers and emit the actions (i.e., faulty control commands) that cause failures on the physical plant. Similar to their approach, we detect/drop faulty control commands if they deviate from the behavioral model of the system. We develop the CPS model as an actor model that contains the progress of time and shows the interactions between the system components. We drive the abstract behavioral model with respect to the trace equivalence that ignores the actions are not observable for the controllers while preserving the actions order in the original model.

Cheng et al. [7] propose a methodology to detect/prevent attacks modifying the data that are used for control decisions in the controllers. These attacks can violate control branches or control integrity (i.e., the number of loop iterations) in the software of the controller. They derive a finite-state automaton (FSA) model in a training phase by monitoring the normal behavior of the program at runtime. They assume sensors data are trustable and therefore they augment the FSA model with the sensors data that report states of the physical plant. In their approach, a controller is considered as a compromised controller when it behaves inconsistently with the corresponding state transitions and the augmented data in the model. In our approach, we perform model checking to generate a model at design time without employing any training phase at runtime. We detect faulty control commands either if they are caused by compromised controllers or the commands are sent based on the receiving false sensor data.

Křikava et al. [17] use data driven models@runtime and create the logic of control systems based on the feedback control loops. They use networks of actors to represent the target system and the adaptation logic. In [3], authors explore the benefits of using the Ptolemy II framework [29] for model-based development of large-scale self-adaptive systems with multiple interacting feedback control loops. They propose a Ptolemy template based on a coordinated actor model to build a self-adaptive system. In their work, model@runtime is used by a coordinator to ensure the satisfaction of the safety properties and to adapt the system by predicting the violation of requirements, e.g. performance degradation. In [33], Sirjani et al. use the Rebeca modeling language to conduct a formal verification of the CPS programs developed in the Lingua Franca language. They study different ways to construct a transition system model for the distributed and concurrent software components of a CPS. They focus on the cyber part and model a faithful interface to the physical part. In our work, we develop executable code that represents system behavior at runtime by mapping the Timed Rebeca to the Lingua Franca, whereas the work in [33] validates a program written in Lingua Franca using model checking.

In [25], authors propose adaptive security policies at runtime. They use ProB model checker to automatically detect the root cause of security violations. They check design models (UML models are transformed to B-method) against secu-

rity constraints at runtime. The authors in [15] propose monitors expressed as automaton models [4] to detect injection attacks against a system. Their automaton models represent parametric specifications to be checked at runtime. Their monitors support event duplication to prevent the system against attacks. They validate the approach by implementing the monitors and performing attack examples on a program taken from the FISSC benchmark [9].

7 Conclusion and Future Work

In this paper, we proposed a method for detecting cyber-attacks on CPSs. In particular, we used a Tiny Twin to detect the attacks on sensors and controllers. We developed an abstraction tool to build the Tiny Twin, which is an abstract version of a state transition system representing the system correct behavior in the absence of an attack. The abstraction tool reduces the transition system based on a list of state variables. The list of state variables includes the variables that store the receiving sensor data and the state variables in the actuators that reflect the changes on the state of the actuators. In our method, we built a monitor module that executes together with the system. It produces an alarm if the sensor data or the control commands are not consistent with the state transitions in the Tiny Twin. We evaluated the capability of our method in detecting attacks on a temperature control system. As the future work, we aim to build a module to recover the system after attacks based on the adaptation plans. We plan to automatically generate potential protection strategies using reinforcement learning.

Acknowledgment

The work of a subset of authors is partly supported by SSF Serendipity project, KKS DPAC Project (Dependable Platforms for Autonomous Systems and Control), and KKS SACSys Synergy project (Safe and Secure Adaptive Collaborative Systems).

References

1. Afra: an integrated environment for modeling and verifying Rebeca family designs. `https://rebeca-lang.org/alltools/Afra` (2021), [Online; accessed Jul 09, 2021]
2. Abera, T., Asokan, N., Davi, L., Ekberg, J.E., Nyman, T., Paverd, A., Sadeghi, A.R., Tsudik, G.: C-flat: control-flow attestation for embedded systems software. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 743–754 (2016)
3. Bagheri, M., Sirjani, M., Khamespanah, E., Khakpour, N., Akkaya, I., Movaghar, A., Lee, E.A.: Coordinated actor model of self-adaptive track-based traffic control systems. Journal of Systems and Software **143**, 116–139 (2018)
4. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: Towards expressive and efficient runtime monitors. In: International Symposium on Formal Methods. pp. 68–84. Springer (2012)

5. Carvalho, L.K., Wu, Y.C., Kwong, R., Lafortune, S.: Detection and mitigation of classes of attacks in supervisory control systems. Automatica **97**, 121–133 (2018)
6. Cheng, B.H., Eder, K.I., Gogolla, M., Grunske, L., Litoiu, M., Müller, H.A., Pelliccione, P., Perini, A., Qureshi, N.A., Rumpe, B., et al.: Using models at runtime to address assurance for self-adaptive systems. In: Models@ run. time, pp. 101–136. Springer (2014)
7. Cheng, L., Tian, K., Yao, D., Sha, L., Beyah, R.A.: Checking is believing: Event-aware program anomaly detection in cyber-physical systems. IEEE Transactions on Dependable and Secure Computing (2019)
8. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS) **31**(3), 1–22 (2013)
9. Dureuil, L., Petiot, G., Potet, M.L., Le, T.H., Crohen, A., de Choudens, P.: Fissc: A fault injection and simulation secure collection. In: International Conference on Computer Safety, Reliability, and Security. pp. 3–11. Springer (2016)
10. Eckhart, M., Ekelhart, A.: A specification-based state replication approach for digital twins. In: Proceedings of the 2018 workshop on cyber-physical systems security and privacy. pp. 36–47 (2018)
11. Gao, C., Seatzu, C., Li, Z., Giua, A.: Multiple attacks detection on discrete event systems. In: 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC). pp. 2352–2357. IEEE (2019)
12. Hewitt, C.: Viewing control structures as patterns of passing messages. Artificial intelligence **8**(3), 323–364 (1977)
13. Jansen, D.N., Groote, J.F., Keiren, J.J., Wijs, A.: 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. pp. 3–20. Springer (2020)
14. Kang, E., Adepu, S., Jackson, D., Mathur, A.P.: Model-based security analysis of a water treatment system. In: Proceedings of Software Engineering for Smart Cyber-Physical Systems. pp. 22–28. ACM (2016)
15. Kassem, A., Falcone, Y.: Detecting fault injection attacks with runtime verification. In: Proceedings of the 3rd ACM Workshop on Software Protection. pp. 65–76 (2019)
16. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Sci. Comput. Program. **98**, 184–204 (2015)
17. Křikava, F., Collet, P., France, R.B.: Actor-based runtime model of adaptable feedback control loops. In: Proceedings of the 7th Workshop on Models@ run. time. pp. 39–44 (2012)
18. Lanotte, R., Merro, M., Munteanu, A.: A process calculus approach to detection and mitigation of plc malware. Theoretical Computer Science **890**, 125–146 (2021)
19. Lee, E., Seo, Y.D., Kim, Y.G.: A cache-based model abstraction and runtime verification for the internet-of-things applications. IEEE Internet of Things Journal **7**(9), 8886–8901 (2020)
20. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security **4**(1), 2–16 (2005)
21. Lima, P.M., Alves, M.V., Carvalho, L.K., Moreira, M.V.: Security against network attacks in supervisory control systems. IFAC-PapersOnLine **50**(1), 12333–12338 (2017)

22. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), 1–27 (2021)
23. Lohstroh, M., Menard, C., Schulz-Rosengarten, A., Weber, M., Castrillon, J., Lee, E.A.: A language for deterministic coordination across multiple timelines. In: 2020 Forum for Specification and Design Languages (FDL). pp. 1–8. IEEE (2020)
24. Lohstroh, M., Romeo, Í.Í., Goens, A., Derler, P., Castrillon, J., Lee, E.A., Sangiovanni-Vincentelli, A.: Reactors: A deterministic model for composable reactive systems. In: Cyber Physical Systems. Model-Based Design, pp. 59–85. Springer (2019)
25. Loulou, H., Saudrais, S., Soubra, H., Larouci, C.: Adapting security policy at runtime for connected autonomous vehicles. In: 2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). pp. 26–31. IEEE (2016)
26. Mitchell, R., Chen, I.R.: A survey of intrusion detection techniques for cyber-physical systems. ACM Computing Surveys (CSUR) **46**(4), 1–29 (2014)
27. Mohan, S., Bak, S., Betti, E., Yun, H., Sha, L., Caccamo, M.: S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In: Proceedings of the 2nd ACM international conference on High confidence networked systems. pp. 65–74 (2013)
28. Moradi, F., Asadollah, S.A., Sedaghatbaf, A., Čaušević, A., Sirjani, M., Talcott, C.: An actor-based approach for security analysis of cyber-physical systems. In: International Conference on Formal Methods for Industrial Critical Systems. pp. 130–147. Springer (2020)
29. Ptolemaeus, C.: System design, modeling, and simulation: using Ptolemy II, vol. 1. Ptolemy. org Berkeley (2014)
30. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using timed Rebeca. Sci. Comput. Program. **89**, 41–68 (2014)
31. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Formal Modeling: Actors, Open Systems, Biological Systems, pp. 20–56. Springer (2011)
32. Sirjani, M., Khamespanah, E.: On time actors. In: Theory and Practice of Formal Methods, pp. 373–392. Springer (2016)
33. Sirjani, M., Lee, E.A., Khamespanah, E.: Verification of cyberphysical systems. Mathematics **8**(7), 1068 (2020)
34. Zhang, Q., Li, Z., Seatzu, C., Giua, A.: Stealthy attacks for partially-observed discrete event systems. In: 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). vol. 1, pp. 1161–1164. IEEE (2018)
35. Zhao, Y., Liu, J., Lee, E.A.: A programming model for time-synchronized distributed real-time systems. In: 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07). pp. 259–268. IEEE (2007)