Journal of Parallel and Distributed Computing 184 (2024) 104780

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Tiny Twins for detecting cyber-attacks at runtime using concise Rebeca time transition system

Fereidoun Moradi *, Bahman Pourvatan, Sara Abbaspour Asadollah, Marjan Sirjani

School of Innovation, Design and Engineering, Mälardalen University, Västerås 722 20, Sweden

ARTICLE INFO

Keywords:
Cyber-security
Intrusion detection systems
Runtime monitoring
Model checking
Model abstraction

ABSTRACT

This paper presents a method for detecting cyber-attacks in cyber-physical systems using a monitor. The method employs an abstract model called Tiny Twin, which is built at design time and is used at runtime to detect inconsistencies. Tiny Twin is a state transition system that represents the observable behavior of the system from the monitor point of view. We model the behavior of the system in the Rebeca modeling language and use Afra model checker to generate the state space. The Tiny Twin is built automatically, by abstracting the state space while keeping the observable actions and preserving the trace equivalence. For doing that we had to solve the complexities in the state space introduced by time-shifts, nondeterministic assignments and abstraction of internal actions. We formally define the state space as Concise Rebeca Timed Transition System (CRTTS), and then map CRTTS to an LTS. The LTS is then fed to a tool to abstract away the non-observable actions.

1. Introduction

CPSs consist of collaborative computational entities that are interacting with physical components through sensors and actuators. Examples of CPSs are water treatment systems, robotic arms, or the power grid. Such CPSs have the combined advantages of the physical and cyber world but are also subject to both threats to safety and security. With increased connectivity, heterogeneous nature, and large-scale deployment, CPSs have larger attack surfaces. Adversaries can manipulate controls or sensor readings through the communications network or tampering the devices, leading to cyber or physical attacks. Examples of such attacks include the security incident at Maroochy Water Services in Australia, which caused a system failure and the release of one million liters of untreated sewage into the river [40]. The attack on the Brazil power grid [8] left nearly 53 million residents in darkness. In security analysis, the investigation of attack schemes often serves as the first step to establishing security in a vulnerable system [12]. In [28], we presented attack schemes on CPSs to discover vulnerabilities, and here, we use an abstract model to build an intrusion detection system for detecting attacks.

Intrusion Detection Systems (IDS) are deployed in communication networks to protect the system against cyber-attacks [26]. Traditional IDSs may struggle to detect complex attacks, but they can be improved by incorporating more advanced logic based on human reasoning, or models of behaviors. Specification-based IDS [26] uses a formal model to monitor the legitimate behavior of a system and detect any deviations from it. This approach generates alarms and performs predefined actions, such as dropping packets, when a violation occurs. The approach presented in this paper can be classified as a specification-based IDS.

In our approach, we detect cyber-attacks on sensor data and control commands using an abstract model called Tiny Twin. This model is built at design time and employed at runtime within a monitor to find inconsistencies. The monitor walks over the Tiny Twin to check whether the sensor data and control commands transmitted in the network are consistent with the state transitions in the Tiny Twin.

In this work, the behavior of the system (and the environment) is modeled using Timed Rebeca modeling language [36], and we perform model checking to verify the requirements and generate the state space. Transitions in the state space may be labeled with actions that are not visible to the monitor (and the controller), the so-called non-observable actions. Therefore, these transitions are abstracted away in the Tiny Twin. If we do not abstract the non-observable actions, the monitoring process may become complex and time-consuming. This is because the monitor would have to perform look-ahead search on multiple branches, which can slow down the monitoring process.

* Corresponding author.
E-mail addresses: fereidoun.moradi@mdu.se (F. Moradi), bahman.pourvatan@mdu.se (B. Pourvatan), sara.abbaspour@mdu.se (S. Abbaspour Asadollah), marjan.sirjani@mdu.se (M. Sirjani).

https://doi.org/10.1016/j.jpdc.2023.104780
Received 10 March 2023; Received in revised form 15 August 2023; Accepted 26 September 2023
Available online 4 October 2023

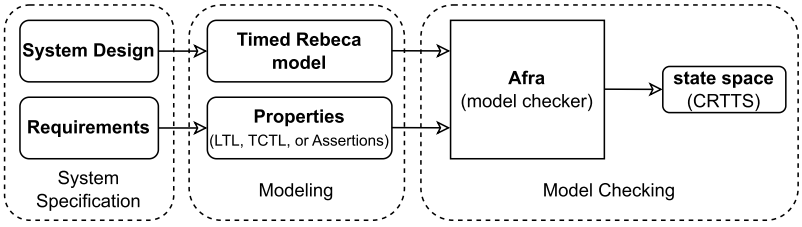


Fig. 1. At design time, the Timed Rebeca model is developed from the specification of the system. The model checking tool of Timed Rebeca, Afra, generates the state space which we formally define as Concise Rebeca Time Transition System (CRTTS).

Similar to [38], we focus on the software aspect of cyber-physical systems. The model of the physical world is abstract, with particular attention to its interface to the software. Including all details of the dynamics of the system is not necessary for our purpose. For example, we simplify the heating and cooling dynamics by considering the average temperature rate instead of modeling heating and cooling in detail. Timed Rebeca [36] allows using nondeterministic assignments for variables. This type of assignment is generally used to model the environment. Nondeterministic assignments create multiple branches in the state space from one state to other states where we have the same nondeterministic assignment as the label on all the transitions, but with different values assigned to the variable in the target state. This phenomenon can also cause inefficiency in runtime monitoring. We resolve this problem in our approach by tagging the transitions with the value of the variable in the target state. At runtime, the value of the variables are determined, and the sequence of actions taken by the system shows a deterministic behavior.

The time in the Timed Rebeca model and subsequently in Tiny Twin is represented as logical time. However, the monitor deals with physical time in the real world, which is based on the physical clocks. We develop the monitor using a coordination language Lingua Franca (LF) [22] to synchronize logical time and physical time. The 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 [24].

Contribution. We develop various techniques and tools to design an efficient and effective IDS. We propose a mapping and abstraction technique to create the Tiny Twin based on a Timed Rebeca model, and we develop a monitor algorithm to detect cyber-attacks using Tiny Twin. We use LF to build an executable model, and for alignment of logical and physical timelines.

We presented our overall approach for detecting attacks in [29]. In this paper, we provide a formal foundation for our approach and present a theory to map the state space of a Timed Rebeca model into a Labeled Transition System (LTS). This is achieved by defining a Concise Rebeca Timed Transition System (CRTTS) and implementing an *ltscast* function to convert CRTTS into an LTS. We use the mCRL2 *ltsconvert* tool [14] to abstract away non-observable actions from LTS while preserving trace equivalence between the original model and its abstracted version.

2. Overview of our approach

We start by developing a Timed Rebeca model from the system specification and verify the requirements using Afra model checker [3] as shown in Fig. 1. The Timed Rebeca model represents the behavior of the system by modeling its components, such as sensors, actuators, and controllers, as actors, and their interactions as message passing [29]. As shown in Fig. 2, we propose a method for abstracting the state space generated by Afra. The state space is mapped into an LTS using our *ltscast* function, and the *ltsconvert* tool [14] is used to abstract the LTS and create the Tiny Twin based on the observable actions in the system. At runtime, the Tiny Twin is used within a *monitor* to detect cyber-attacks on sensor data and control commands as shown in Fig. 3. To prevent damage to the system, the *monitor* drops control commands that are not consistent with the state transitions in the Tiny Twin.

We consider a physically secure *monitor* that is connected to the controller via secure channels. When a controller is compromised, the best way to address incorrect behavior is by introducing a physically independent secure proxy, as suggested by McLaughlin and Mohan et al. [25,27]. This proxy remains disconnected from the Internet or USB and is securely connected to the controller. Our monitor functions as a proxy, responsible for ensuring the correct behavior of the system, while the controller handles the crucial task of controlling the physical process, which involves potentially risky communications through the Internet. Therefore, any upgrades or modifications to the control system will be applied to the controller, not the secure proxy.

Threat Model. We assume an attacker can inject arbitrary code into the controller software (e.g., the firmware of PLCs) and as a consequence, can drop actuator commands, read/modify sensor data coming from the plant, and manipulate communications between the controllers. This means that the attacker can modify the behavior of the system and disrupt the physical process through the false data injection attack, which uses forged sensor data to cause harmful control decisions. However, the assumed attacker can not maliciously alter sensor signals at a network level, or within the sensor devices, because the attacker is assumed to reside in the controller only. The coordinated attack is also possible, where one or more of the above attacks are performed together in an attempt to disrupt the correct functionality of the system. According to [28], we simulate these attacks on sensor data and control commands via compromised components.

The standard semantics of Timed Rebeca in terms of timed transition system with discrete progress of time steps, called TTS, that is proposed in [17]. In TTS, each statement of a message server is executed at a time, and the execution of different message servers is interleaved. In this paper, we consider the state space generated by Afra based on the TTS semantics of Timed Rebeca models.

Timed models result in an infinite number of states in the state space due to the progress of time, making the transition systems unbounded. To overcome this, a new notion of equivalence between two states called the *shift-equivalence relation* is introduced for Timed Rebeca in [17] and [16]. Afra uses the *shift-equivalence relation* to generate the state space, but the state space generated by Afra is not formally defined. In order to define our abstraction technique, we first define Bounded Rebeca Timed Transition System (BRTTS) which is the result of applying the *shift-equivalence relation* between the states in TTS. We then abstract the internal actions in BRTTS to derive Concise Rebeca Timed Transition System (CRTTS).

3. Background: Timed Rebeca and Lingua Franca

In this section, we provide an overview on Timed Rebeca [36], and describe Lingua Franca programming language [23]. Then, we present a running example that is used throughout the paper to describe our method.

Timed Rebeca. Rebeca [35] is an actor-based language for modeling and formal verification of concurrent and distributed systems. Actors, called *rebecs*, are instances of *reactive classes* and communicate via asyn-

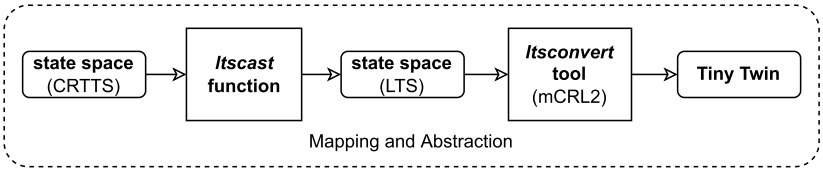


Fig. 2. We define the *ltscast* function to map CRTTS into LTS. The LTS is then abstracted by the mCRL2 *ltsconvert* tool to create the Tiny Twin.

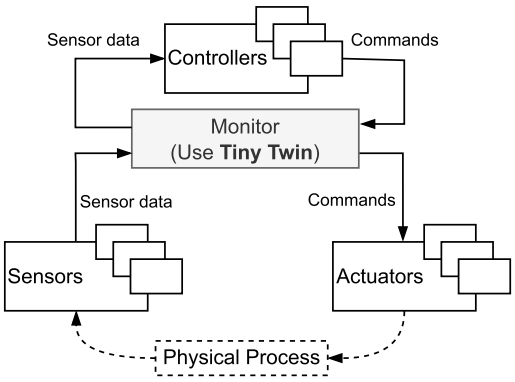


Fig. 3. At runtime, the Tiny Twin is used within the *monitor* to detect cyber-attacks on sensor data and control commands. The monitor drops control commands that are not consistent with the state transitions in the Tiny Twin.

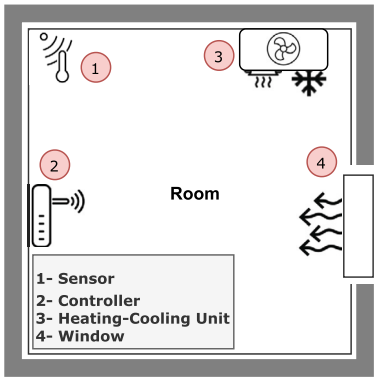


Fig. 4. The components of the temperature control system and its environment.

chronous 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 which 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 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 [3]. 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, 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 [16] 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 all should have the same difference in their amount.

***Lingua Franca* (LF)**. Lingua Franca is a meta language based on the Reactor model for programming CPS [22,24]. 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 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 [13], 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 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 timer triggers at the start time of the execution), and then triggers every *period*. LF 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 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 [38,37].

Running Example: A Simplified Temperature Control System.

Fig. 4 shows the components of the system and its environment, and its Timed Rebeca model and LF code are shown in Listing 1. We use the transition system of this simplified version to illustrate the mapping from CRTTS to LTS, and generate the Tiny Twin (see Section 4).

The temperature control system is designed to maintain the temperature of a room within a desired range. There is a window inside the room and the outside air blows inside when the window is open. The temperature of the room is slowly affected by outside air blowing, whether the outside weather is colder or warmer than the current temperature of the room. In the system, the controller receives sensor data from the sensor and activates the cooling or heating process, or switch the heating/cooling process off. In Timed Rebeca, we abstract the data values and represent them in a symbolic way, for example, the values between 26.55 to 27.55 are represented as 27.

Listing 1(a) shows the Timed Rebeca model that has four reactive classes: controller, sensor, hc_unit and room. In the reactive class controller, the message server getsense(int t) takes temperature value and sends the message activateh() or switchoff() to the corresponding message servers in the reactive class hc_unit depending on the temperature value (line 6 to 16). We model the environment using the reactive class room. The change in the temperature of the room is modeled using a nondeterministic assignment of values 0 or 1 to the variable outside_air_blowing

```
1  reactiveclass Controller(8){
2      knownrebecs{ HC_Unit hc_unit;}
3      statevars{ boolean h_active; int temperature;}
4      Controller(){
5          h_active = false; temperature = 20;}
6      msgsrv getsense(int t){
7          temperature = t;
8          if (21 > temperature && h_active == false) {
9              hc_unit.activateh();
10             h_active = true;
11         } else
12         if (21 <= temperature  && h_active == true){
13             hc_unit.switchoff();
14             h_active = false;
15         }
16     }
17 }
18 reactiveclass Sensor(5){
19     knownrebecs{ Room room; Controller controller; }
20     msgsrv getTemp(int temp) {
21         controller.getSense(temp);}
22 }
23 reactiveclass HC_Unit(5){
24     knownrebecs{ Room room; }
25     statevars{ boolean heater_on; }
26     HC_Unit(){ heater_on = false; }
27     msgsrv activateh(){
28         room.regulate(1); heater_on = true;}
29     msgsrv switchoff(){
30         room.regulate(0); heater_on = false;}
31 }
32 reactiveclass Room(5){
33     knownrebecs{ Sensor sensor; }
34     statevars{
35         int temp, outside_air_blowing;
36         int regulation; }
37     Room(){ temp = 21; regulation = 0;
38         outside_air_blowing = 0;
39         self.tempchange();
40     }
41     msgsrv tempchange() {
42         outside_air_blowing = ?(1,0);
43         temp = temp - outside_air_blowing + regulation;
44         sensor.getTemp(temp);
45         self.tempchange() after(10);
46     }
47     msgsrv regulate(int v) { regulation = v;}
48 }
49 main{
50     Controller controller(hc_unit):();
51     Sensor sensor(room,controller):();
52     HC_Unit hc_unit(room):();
53     Room room(sensor):(); }
```

(a) Timed Rebeca model

```
1  target Cpp {fast: false};
2  reactor Controller { input getsense:int;
3      output activateh:int; output switchoff:int;
4      state h_active:bool(false);
5      reaction(getsense) -> activateh, switchoff {=
6       int temperature = *getsense.get();
7       if(21 > temperature && h_active == false) {
8          activateh.set(1);
9          h_active = true;
10      } else
11      if(*21 <= temperature && h_active == true) {
12         switchoff.set(0);
13         h_active = false;
14      }
15     =}
16 }
17 reactor Sensor {
18     input getTemp:int; output out:int;
19     reaction(getTemp) -> out {=
20         out.set(getTemp.get());  =}
21 }
22 reactor HC_Unit {
23     input activateh:int; input switchoff:int;
24     output regulate:int;
25     reaction(activateh) -> regulate {=
26         regulate.set(1); =}
27     reaction(switchoff) -> regulate {=
28         regulate.set(0); =}
29 }
30 reactor Room {
31     input regulate:int; output getTemp:int;
32     state temperature:int(21);
33     state cold_air_blowing:int(0);
34     state regulation:int(0);
35     timer start(0, 10 sec);
36     reaction(start) -> getTemp {=
37         cold_air_blowing = rand() % 3 + (-1);
38         temp = temp - cold_air_blowing + regulation;
39         getTemp.set(temp);
40     =}
41     reaction(regulate) {=
42         regulation = *regulate.get();
43     =}
44 }
45 main reactor Simple_Temperature_Control_System {
46         room = new Room(); sensor = new Sensor();
47         unit = new HC_Unit();
48         controller = new Controller();
49         room.getTemp -> sensor.getTemp;
50         sensor.out -> controller.getsense;
51         unit.regulate -> room.regulate;
52         controller.activateh -> unit.activateh;
53         controller.switchoff -> unit.switchoff;}
```

(b) LF code

Listing 1: (a) The Timed Rebeca model and (b) the LF code of the simplified temperature control system example.

(line 42). While the heating and cooling process is continuous in nature, we model the state changes in a discrete way. We model the changes at certain times (or when an event occurs). We model the change in the temperature of the room in a periodic way, every 10 units of time (line 45). In the main section, we create instances of the four reactive classes (line 49 to 53). Similar to the Timed Rebeca model of the system, the LF code implements all components of the system as shown in Listing 1(b). The input port getsense in the reactor controller is defined to get temperature value (line 2), and the output ports activateh and switchoff are defined to send commands to the hc_unit (lines 8 and 12). We set the value of activateh to 1 to trigger the heating, the value of switchoff to 0 to trigger the switch off in the hc_unit. A key property of LF is the logical time. All events occur at an instant in logical time. We use a timer (line 35) to periodically invoke the reactions and model the periodic events (similar to after in Timed Rebeca model).

4. From the state space to the Tiny Twin

We create the Tiny Twin based on the state space (generated by Afra) by keeping only the observable actions for the monitor while abstracting away the remaining actions. Here, we introduce a Timed Rebeca model and its constituents based on the definition of Timed Rebeca semantics [14,17]. We refer to the Timed Transition System semantics of Timed Rebeca as RTTS. In Section 4.2, we define Bounded RTTS (BRTTS) using *shift-equivalence relation* and abstract the internal actions to derive Concise RTTS (CRTTS). Internal actions in BRTTS are caused by executing statements of a message server in the Timed Rebeca model [17]. In Section 4.3, we define the *ltscast* function to map CRTTS into LTS, which we can then use in *ltsconvert* tool to abstract the non-observable actions. Finally, we explain the algorithm for the *ltscast* function in Section 4.4.

The original state space generated by Afra is deterministic. When the original state space is deterministic, then bisimulation and trace equivalence coincide.

4.1. Afra state space

A Timed Rebeca model can be used to model concurrent, distributed and cyber-physical systems and is defined as follows.

Definition 1. ***Timed Rebeca model*** [17]. A Timed Rebeca model M is a set of rebecs. Each rebec $r \in M$ is defined as a tuple $r = (\mathcal{V}_r, \mathcal{M}_r, \mathcal{K}_r)$ where $\mathcal{V}_r$ is defined as the local state of the rebec, $\mathcal{M}_r$ is the set of its message servers, and $\mathcal{K}_r$ is the set of its known rebecs.[1] The local state $\mathcal{V}_r$ of rebec r is a tuple of (V_r, B_r, pc_r, res_r), where V_r is the set of variables of rebec r together with their values, B_r is the message bag of rebec r, $pc_r \in \mathbb{N} \cup \{null\}$ is the program counter which points to the statement in the body of the current message server[2] ($null$ if r is idle), and $res_r \in \mathbb{N}_0$ ($\mathbb{N}_0$ is the set of non-negative integers) is the resuming time, $res_r > 0$ if rebec r is executing a delay. □

In a Timed Rebeca model, rebecs can be used to represent components of a CPS or a distributed system. The interactions between rebecs are modeled as message passing. The messages can also be seen as events that trigger the execution of a message server that can be seen as an event handler. A Timed Rebeca message is defined as follows.

Definition 2. ***Timed Rebeca*** **Message**. A message in Timed Rebeca model is defined as $tmsg = ((sid, rid, mid, ps), ar, dl)$, where sid and rid are the name of the sender and receiver rebecs of the message; respectively, mid is the name of the message server in Timed Rebeca model, ps is the set of input parameters, $ar \in \mathbb{N}_0$ and $dl \in \mathbb{N}_0$ are the arrival time and deadline of the message, respectively. □

The RTTS describes the behavior of a Rebeca model. It consists of a finite set of states (each state has the current time *now*), a finite set of actions, and a transition relation between states. Before defining the formal semantics of Timed Rebeca as an RTTS, the actions in RTTS are defined by the following definition.

Definition 3. ***Actions in Rebeca Timed Transition System***. There are three types of actions in Rebeca Timed Transition Systems which is defined below. The condition for triggering of each action is also explained:

1. Take Message: When a rebec r is idle ($pc_r = null$) and its message bag is not empty ($B_r \neq \emptyset$), it takes a Timed Rebeca message $tmsg$ for execution from its message bag if arrival time of $tmsg$ is less than or equal to the current time $'now'$ ($tmsg.ar_r \leq now$).

2. Internal Action: The execution of a statement in the message server m of a rebec r is an internal action that is shown by $\tau.r.m$. This action shows the changes of local state of the rebec r. The internal action is taken in state s, if there is a rebec r where $pc_{s,r} \neq null$ and $res_{s,r} = now_s$ (the value of $res_{s,r}$ does not change during the execution of the statement, except for running a delay statement).

3. Time Progress: Time Progress shows progress of $n \in \mathbb{N}$ units of time. Time progress is triggered when there is no internal or take message actions. Hence, the set of actions is $Act = \{tmsg | \ tmsg \in B_r \wedge r \in M \wedge tmsg.ar_r \leq now\} \cup \{\tau.r.m | r \in M \wedge m \in \mathcal{M}_r\} \cup \mathbb{N}$. □

[1] A known rebec for a rebec r is a rebec which can be a receiver rebec of the message msg that is sent by the sender rebec r.

[2] The current message server is the message server that is executed and has not finished yet.

We show the diagram of the transition system of the running example to provide insight into the system model before presenting the formal semantics of Timed Rebeca in RTTS.

Running Example: Afra state space of the simplified temperature control system. Fig. 5(a) shows the state space generated by Afra for the Timed Rebeca model in Listing 1(a). This example is a reactive system, and its diagram shows that it exhibits recurrent behavior. Each state has a state variable temp that its value is changed when the action tempchange is executed (see line 43 in Listing 1(a)). The time+ = 10 denotes that the logical time progresses by 10 units of time (see line 45 and the red color transitions in the diagram). The gettemp, getsense, tempchange, regulate, activateh, and switchoff are take message actions which are shown on transitions. The states with multiple outgoing transitions (in gray color) are due to the nondeterministic assignment for the variable outside_air_blowing in the Timed Rebeca model (see line 42). In the state space generated by Afra, the internal actions correspond to the changes on the program counter are abstracted. We explain about these internal actions in the Definition 8.

In the following, we define RTTS as the formal semantics of a Timed Rebeca model.

Definition 4. ***Formal Semantics of Timed Rebeca in RTTS***. RTTS of a Timed Rebeca model M is a tuple of $(S, s_0, Act, \longrightarrow)$, where S is the set of states, $s_0 \in S$ is the initial state, Act is the set of actions defined in Definition 3, and $\longrightarrow \subseteq S \times Act \times S$ is the transition relation set.

– **States.** Each state $q \in S$ has a state descriptor (global state) which consists of the local states of all rebecs in the model M, together with the current time at the state q that is $now_q \in \mathbb{N}_0$. The local state of rebec r in state q is shown by $(V_{r,q}, B_{r,q}, pc_{r,q}, res_{r,q})$. So, the state descriptor of state q is defined as $\{(V_{q,r}, B_{q,r}, pc_{q,r}, res_{q,r}) | r \in M\} \cup \{now_q\}$, where M is the Timed Rebeca model in Definition 1.

– **Transition Relations.** For each transition relation $(s, act, t) \in \longrightarrow$, the state descriptor of state t is obtained by the following rules. Note that the conditions to take each action is defined in Definition 3:

1. act is a take message: After executing the act, $pc_{t,r}$ is set to the first statement of the message server corresponding to $tmsg$, and $res_{t,r}$ is set to now_t (which is the same as now_s). Note that the other members of the state descriptor of state t remain the same as in the state s.

2. act is an internal action: The statement of message server of rebec r specified by $pc_{s,r}$ is executed and one of the following cases occurs based on the type of the statement.

a) Non-delay statement: the execution of such a statement may change the value of a variable of rebec r, or send a message to another rebec, in this case, the message is added to the message bag of the receiver rebec in $\mathcal{K}_r$ (the set of known rebecs of rebec r). Here, $pc_{t,i}$ is set to the next statement (or $null$ if there are no more statements).

b) Delay statement with parameter $d \in \mathbb{N}$: the execution of a delay statement sets $res_{t,r}$ to $now_s + d$. All other elements of the state remain unchanged.

3. act is a time progress: In this case, now_t is set to $now_s + n$ where n is the minimum value which makes one of the aforementioned conditions become true. For any rebec r, if $pc_{s,r} \neq null$ and $res_{s,r} = now_t$ (the current value of $pc_{s,r}$ points to a delay statement), $pc_{t,r}$ is set to the next statement (or to $null$ if there are no more statements). □

Reactive systems are known for their never-ending processes. Progress of time and increasing the value of the current time can create an infinite state space. In the next subsection, we explain how this problem is addressed.

4.2. Defining BRTTS based on shift-equivalence relation

The progress of time results in an infinite number of states in RTTS of Timed Rebeca models. These models are used to represent reactive systems that generally show periodic or recurrent behaviors, meaning they perform periodic actions over an infinite period of time. Therefore,

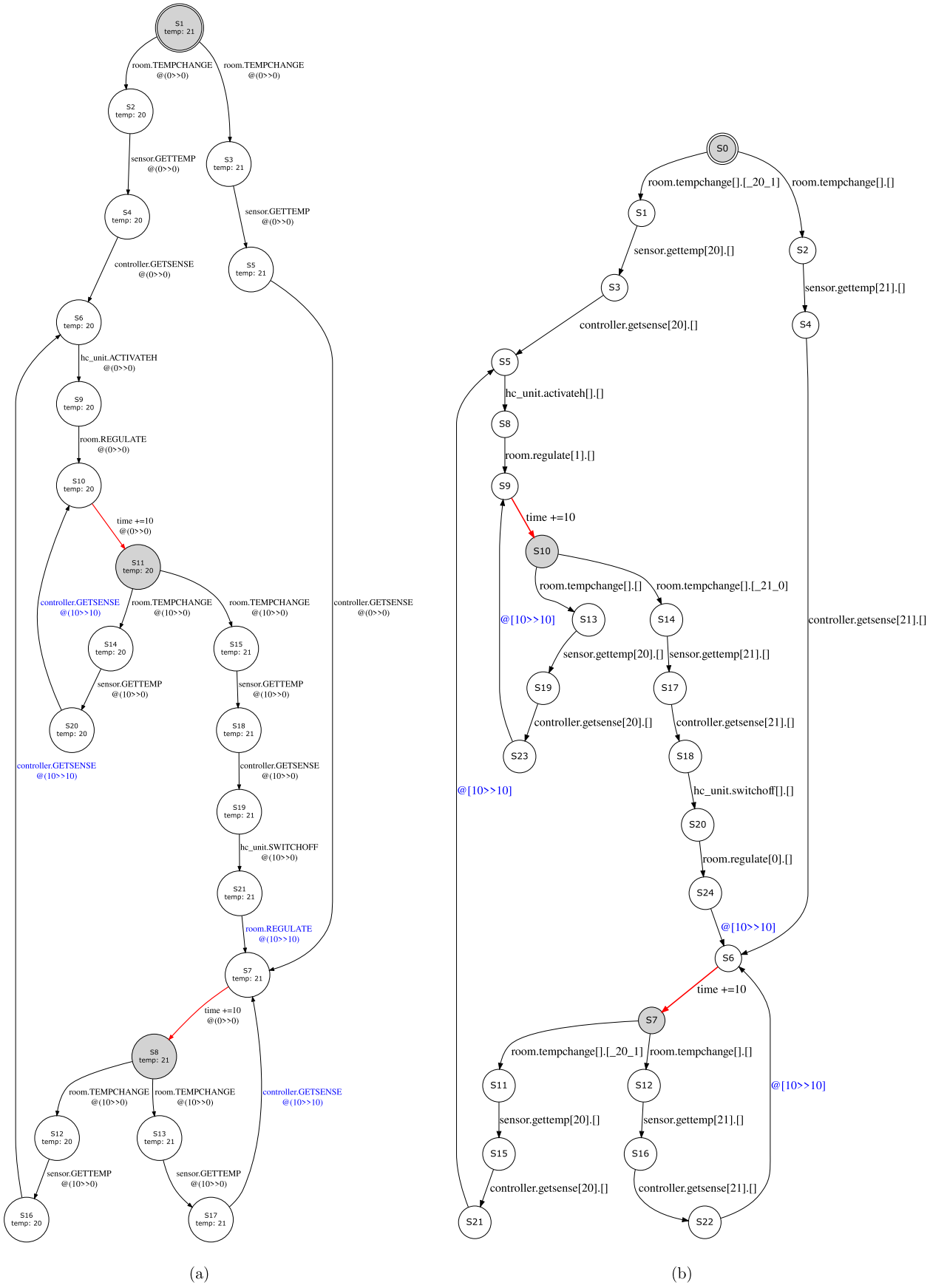


Fig. 5. (a) The state space of the Timed Rebeca model of the simplified temperature control system in Listing 1, (b) the LTS created by the *ltscast* function for the simplified temperature control system.

to make the transition systems bounded, a new notion of equivalence between two states called *shift-equivalence relation* is proposed in [16].

Intuitively, in *shift-equivalence relation*, two states of a RTTS are equivalent if and only if they are the same in all of their parts in the state descriptors except for those parts related to the time (the value of now in the state, the resuming time res_r for local state of each rebec r in the state, arrival time ar and deadline dl of all messages in all message bags for all rebecs in the state). Therefore, shifting the times of those parts in one state makes it the same as the other state [16].

Definition 5. ***Shift-Equivalence Relation in RTTS***. In a $RTTS = (S, s_0, Act, \longrightarrow)$, two states $s, t \in S$ are in shift-equivalence relation $s \cong t$ if and only if each element of two states that are related to time has time difference $\Delta \in \mathbb{N}_0$. For the state descriptors of states s and t the following conditions hold.
1. $now_s = now_t + \Delta$
2. M *is a Timed Rebeca model and* $\forall r \in M$, *there exist* $V_{r,s} = V_{r,t}$, $\| B_{r,s} \| = \| B_{r,t} \|$ (size of two bags are equal), $pc_{r,s} = pc_{r,t}$, $res_{r,s} = res_{r,t} + \Delta$
3. $\forall tmsg \in B_{r,s}$ *exists* $tmsg' \in B_{r,t}$ *where* $tmsg.sid = tmsg'.sid$, $tmsg.rid = tmsg'.rid$, $tmsg.mid = tmsg'.mid$, $tmsg.ps = tmsg'.ps$, $tmsg.ar = tmsg'.ar + \Delta$, $tmsg.dl = tmsg'.dl + \Delta$. □

We define a function to obtain the time difference value between two shift-equivalent states, and we use this function in the definition of BRTTS.

Definition 6. Shift function. The function $\delta : S \times S \mapsto \mathbb{N}$ receives two states that are shift-equivalent, and returns a positive integer which is equal to the Δ defined in Definition 5.

The BRTTS in which the transition system is bounded is defined as follows.

Definition 7. ***Bounded Rebeca TTS*** (***BRTTS***). Let $A = (S, s_0, Act, \longrightarrow)$ be a RTTS. BRTTS A is $A' = (S', s_0, Act', \longrightarrow')$, where:
States. The set of states of BRTTS is the subset of states of RTTS where the states which are in shift-equivalence relation are merged: $S' = S \setminus \{t | \exists s \in S'; s \cong t \wedge now_s < now_t\}$
Actions. $Act' = Act \times \mathbb{N}_0$ where Act is the set of actions in RTTS in Definition 3.
Transition Relations. Transitions in BRTTS are the same transitions in RTTS except for the transitions leading to or from states that have the shift-equivalent state in BRTTS. One of the following conditions holds.

1. $\dfrac{q \xrightarrow{act} p \ \wedge \ p,q \in S'}{(act,0) \in Act',\ q \xrightarrow{(act,0)}' p}$

2. $\dfrac{q \xrightarrow{act} p \ \wedge \ q \notin S' \ \wedge \ p \in S' \ \wedge \ s \in S' \ \wedge \ s \cong q}{(act,\delta(s,q)) \in Act',\ s \xrightarrow{(act,\delta(s,q))}' p}$

3. $\dfrac{q \xrightarrow{act} p \ \wedge \ q \in S' \ \wedge \ p \notin S' \ \wedge \ s \in S' \ \wedge \ s \cong p}{(act,\delta(s,p)) \in Act',\ q \xrightarrow{(act,\delta(s,p))}' s}$

In BRTTS, the execution of statements in a message server of a rebec (after executing a take message action) is shown as a sequence of internal actions. In an internal action path, all states with only one incoming transition and one outgoing transition are merged. The states with branches are not merged in the model. This reduction is based on the (weak) bisimulation equivalency [17]. The definition of BRTTS becomes concise, CRTTS, by merging all states between internal actions in the internal action paths of BRTTS. We define an internal action path as follows.

Definition 8. Internal Action Path. Let $(S, s_0, Act, \longrightarrow)$ be a BRTTS. An internal action path is a longest sequence of transitions from state s to state t, where the first transition is a take message action and the rest transitions are internal action τ, and all states between state s and t have only one incoming transition and one outgoing transition. Internal action path from state s to state t is denoted as $P_{s,t} = s \xrightarrow{(act,\alpha_0)} s_1 \xrightarrow{(\tau.r.m,\alpha_1)} s_2 \cdots s_n \xrightarrow{(\tau.r.m,\alpha_n)} t$ where act is a take message action, $r = act.rid$, and $m = act.mid$. □

An example of an internal action path in an example BRTTS is shown in Fig. 6. The transitions from state S2 to state S7 represent the internal actions where they are merged with state S2 and make the state space concise. Based on the definition of internal action path, we define CRTTS as follows.

Definition 9. ***Concise RTTS*** (***CRTTS***). Let $A = (S, s_0, Act, \longrightarrow)$ be a BRTTS. CRTTS A is $A' = (S', s_0, Act', \longrightarrow')$, such that for each internal action path P_s in A there exists a transition $s \xrightarrow{(act,\Sigma_{i=0}^{n}\alpha_i)} t$ in A'. The following condition is held for the internal action paths.

$$\frac{P_{s,t} = s \xrightarrow{(act,\alpha_0)} s_1 \xrightarrow{(\tau.r.m,\alpha_1)} s_2 \cdots s_n \xrightarrow{(\tau.r.m,\alpha_n)} t \in A}{s \xrightarrow{(act,\Sigma_{i=0}^{n}\alpha_i)}' t, \ \ s_i \notin S' \ \ i \in [1..n]}$$

where act is a take message action, $r = act.rid$, and $m = act.mid$. □

4.3. Mapping CRTTS to LTS

There are two issues in CRTTS where we need to apply changes to obtain LTS.
1. For each transition $s \xrightarrow{(act,\alpha)} t$ where $\alpha > 0$, the action act comes with the time shifting value α. There are cases where act is a non-observable action that must be abstracted from the behavioral model, as an example see transition (act, α) from S8 to S1 in Fig. 6(b)). But we do not want to abstract the time shift and its value. In this case, we split the transition $s \xrightarrow{(act,\alpha)} t$ into two transitions $s \xrightarrow{act} t_1$ and $t_1 \xrightarrow{\alpha} t$ (see transition act between S8 and S10, and transition α between S10 and S1 in Fig. 6(c)). This way we can keep the time shift while abstracting the non-observable action.
2. Nondeterministic assignments are implemented in the model checker where the state space is generated but they are not defined in the formal semantics of Timed Rebeca [36]. Nondeterministic assignments create multiple branches in the state space and are shown with the same label on all the transitions where different values are assigned to the variable in the target states. In case of nondeterministic assignments, there may be two transitions $s \xrightarrow{act} t_1$ and $s \xrightarrow{act} t_2$ where $t_1 \neq t_2$ (see outgoing transitions at S2 in Fig. 6(b)). We use the different state descriptors t_1 and t_2 and change the actions in the transitions to be $s \xrightarrow{act.v_1} t_1$ and $s \xrightarrow{act.v_2} t_2$ where v_1 and v_2 are different values for a state variable in the states t_1 and t_2 (see outgoing transitions at S2 in Fig. 6(c)).

The target of our mapping which is the input to the mCRL2 *ltsconvert* tool is an LTS. A labeled transition system (LTS) is defined as follows [14].

Definition 10. ***LTS***. LTS associated with the state space of a mCRL2 model is a tuple of $(S, s_0, Act, \longrightarrow)$, where:
States. S is the set of states.
Initial State. s_0 is the initial state.
Actions. Act is a finite set of actions including the internal action τ.
Transition Relations. $\longrightarrow \subseteq S \times Act \times S$ is the transition relation. □

We define the *ltscast* function to map CRTTS to LTS as follows.

Definition 11. ***LTS cast Function*** (***ltscast***). Every CRTTS $A = \langle S, s_0, Act, \longrightarrow \rangle$ can be mapped to LTS $L = \langle S', s'_0, Act', \longrightarrow' \rangle$ by LTS cast Function $ltscast$: CRTTS $\longrightarrow$ LTS, where
– For all $s \in S$, there exists $s' \in S'$,
– $Act = Act'$, and

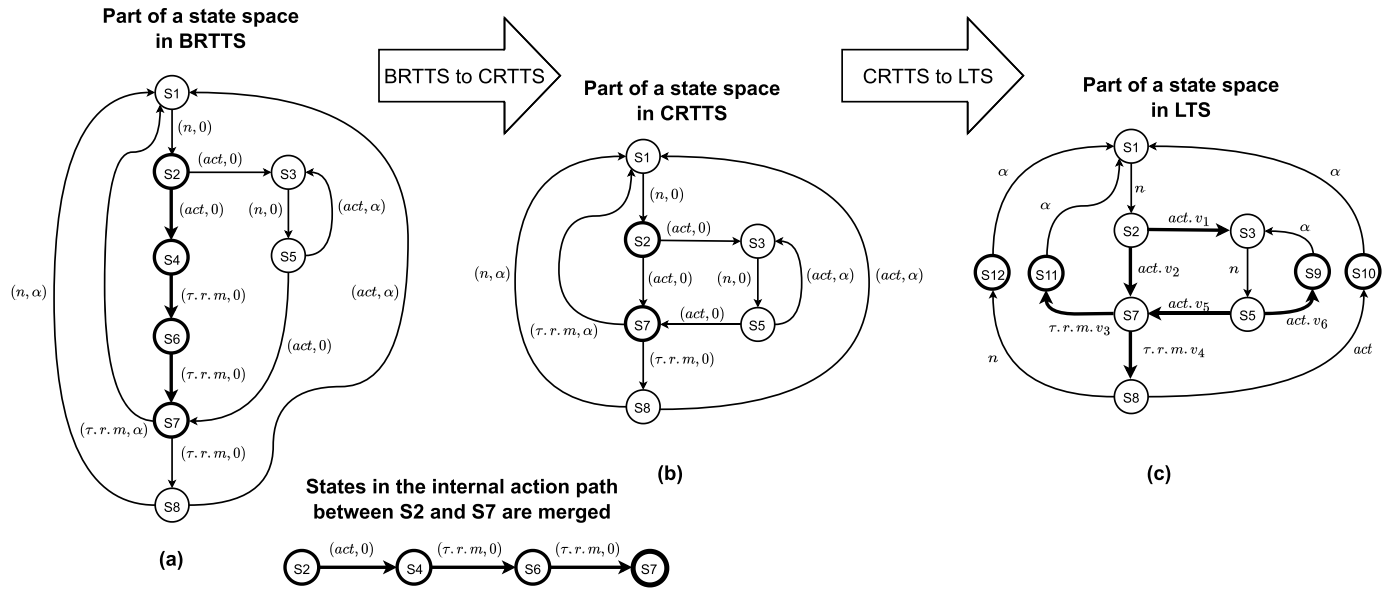


Fig. 6. (a) An example of a state transition system in BRTTS. There is an internal action path between two states S2 and S7 in BRTTS. (b) The CRTTS is created by merging states S4 and S6 with S2 in the internal action path. (c) The transitions with time shifting value $\alpha > 0$ outgoing from states S5, S7, and S8 are split into two subsequent transitions, and the nondeterministic transitions outgoing from states S2, S5 and S7 are labeled with different state descriptors in LTS.

– $\longrightarrow' \subseteq S' \times Act' \times S'$ is defined as a transition relation if and only if one of the following conditions holds.

1. $\dfrac{s \xrightarrow{(act,\alpha)} t \wedge (\alpha \neq 0)}{s \xrightarrow{act} t_1, t_1 \xrightarrow{\alpha} t, S' = S \cup \{t_1\}}$ 2. $\dfrac{s \xrightarrow{(act,\alpha)} t \wedge (\alpha = 0)}{s \xrightarrow{act} t}$ 3. $\dfrac{s \xrightarrow{act} t_1 \wedge s \xrightarrow{act} t_2 \wedge t_1 \neq t_2}{s \xrightarrow{act.v_1} t_1 \wedge s \xrightarrow{act.v_2} t_2}$

where v_1 and v_2 are two different values for a state variable in the states t_1 and t_2. □

The example CRTTS in Fig. 6(b) shows that the transitions (act, α) and (n, α) from S8 to S1, and from S5 to S3, and the transition $(\tau.r.m, \alpha)$ from S7 to S1 come with the time shifting value $\alpha > 0$. Therefore, these transitions respectively are split into two transitions act and α, and $\tau.r.m$ and α in Fig. 6(c). Moreover, the outgoing transitions from states S2, S5 and S7 are labeled with different values v_1, v_2, v_3, v_4, v_5 and v_6 for a state variable in the state descriptors of the target states.

4.4. ltscast function

The *ltscast* function gets a CRTTS as input (the state space generated by Afra) and produces an LTS as output (the input model for mCRL2 *ltsconvert* tool). The function follows two steps to create an LTS (see the details of the *ltscst* algorithm in Appendix A). (1) It traverses transitions one by one and divides transitions into two subsequent transitions if they have a non-zero value for the time shifting. One subsequent transition represents an action, and the other transition represents a time shifting. (2) It traverses transitions and tags transitions that introduce nondeterminism with different values to remove nondeterminism, these values are the values of a state variable in the target states of the transitions.

The mapping algorithm traverses the state space using a Depth-First search to visit all states and transitions. The time complexity of the algorithm is $O(\max(V, E))$, and its space complexity is $O(V + E)$, where V is the number of states and E is the number of transitions in the state space. The relation between the CRTTS and the created LTS is bisimulation.

Running Example: Mapping CRTTS to LTS and Creating the Tiny Twin. We use *ltscast* to map the state space generated by Afra and create an LTS (see Fig. 5 (b)). The transition between two states that are in the *shift-equivalence relation* is tagged with the value of the time-shift (in blue color). The time shifting notation $[a \gg b]$ on state transitions denotes that the source state has the current time value a (i.e., the value of variable *now*), which is shifted by the value b and becomes the time value at the target state (together with all the other elements related to time).

The *ltsconvert* tool abstracts the LTS to its equivalence relation. To create the Tiny Twin of the temperature control system, the modeler provides the tool with a list of labels that denote the silent transitions. The silent transitions are abstracted away from the LTS while preserving trace equivalence. In this system, the actions getsense, activateh and switchoff are observable in the system behavior from the controller point of view, while actions tempchange, gettemp, regulate and tempchange are non-observable. We may want to check properties on the control system, such as "the command switchoff will be issued in less than 10 units of time if temp = 21" or "the command activateh will be issued if temp = 20 and the action switchoff has already executed". The *ltsconvert* tool receives the labels of non-observable actions as a list of silent transitions to abstract the LTS. The abstraction is applied by the tool, and the resulting abstract model (i.e., Tiny Twin) is shown in Fig. 7. The Tiny Twin has 10 states and 13 transitions, while the original CRTSS in Fig. 5(a) has 21 states and 25 transitions. In this example, because the system is simplified and does not have any complex behavior, we do not expect to see a significant reduction in the number of states and transitions compared to the original model.

5. Monitor algorithm

The monitor observes sensor data and control commands as events. It checks if the events are consistent with the transitions in the Tiny Twin. The monitor must traverse all time shifting and internal actions at each state during monitoring until it finds the event as the label on the outgoing transition. The Tiny Twin has transitions of four types: take message, time progress, time shifting, and internal action (see subsection 11). Although we abstract internal actions in the internal action paths (see Definition 8), the internal actions that correspond to the observable actions and involve delay statements in the Time Rebeca model are still in the model. If the monitor gets an event which cannot be matched with the transitions in the Tiny Twin, then it shows a mismatch between what the monitor expects and what is happening, and

the monitor makes an alarm and terminates the monitoring (see the details of the *monitor* algorithm in Listing 1).

Algorithm 1: Monitor Algorithm.

```
   Input: an LTS (S' : stateSet, s'_0 : initalState, Act' : actionSet, T' : transitionSet)
   Output: an alarm (Time, Event)
1  begin
2      S = {(s'_0, 0, getTSet(s_0))};
3      proceed ← true;
4      while proceed do
5          (e, k) ← getEvent();
6          proceed ← false;
7          chk ← true;
8          while st ∈ S & chk & (getTMsg(st, e) || getTTime(st, k)) do
9              t ← getTMsg(st, e);
10             chk ← false;
11             if t! = null then
12                 proceed ← true;
13                 chk ← true;
14                 st.transSet = st.transSet \ t;
15                 if st.transSet == ∅ then
16                     S = S \ st;
17                 S = S ∪ {(t.target, st.now, getTSet(t.target))};
                   put: e;
18             t ← getTTime(st, k);
19             if t! = null & (k >= st.now + t.getTime()) then
20                 proceed ← true;
21                 chk ← true;
22                 st.transSet = st.transSet \ t;
23                 if st.transSet == ∅ then
24                     S = S \ st;
25                 S = S ∪ {(t.target, st.now + t.getTime(), getTSet(t.target))};
26         chk ← true;
27         while st ∈ S & chk & (getTTau(st) || getTTime(st, k)) do
28             chk ← false;
29             t ← getTTau(st);
30             if t! = null then
31                 proceed ← true;
32                 chk ← true;
33                 st.transSet = st.transSet \ t;
34                 if st.transSet == ∅ then
35                     S = S \ st;
36                 S = S ∪ {(t.target, st.now, getTSet(t.target))};
37             t ← getTTime(st, k);
38             if t! = null & (k >= st.now + t.getTime()) then
39                 proceed ← true;
40                 chk ← true;
41                 st.transSet = st.transSet \ t;
42                 if st.transSet == ∅ then
43                     S = S \ st;
44                 S = S ∪ {(t.target, st.now + t.getTime(), getTSet(t.target))};
45     return new alarm(k, e);
```

In the following, we explain the details of the monitor algorithm where the Tiny Twin in the form of LTS (S', s'_0, Act', T') is the input, and where S', s'_0, Act' and T' are the set of states, the initial state, the action set, and the transition set, respectively. The algorithm starts from the initial state of the Tiny Twin, $s'_0 \in S'$, as the only state in the set of current states, S. The set S keeps all current states during monitoring along with the logical time and the outgoing transitions of the current states (line 2). The algorithm begins and proceeds its operation by observing an event (a sensor data or a control commands) at the current logical time k (line 5). The algorithm checks the outgoing transitions at each state in the set S, and compares the observed event e to the labels on the outgoing transitions, and traverses the transition that matches the event using the function getTMsg (line 9 to 17). It also checks if the time k is one of the outgoing transitions at the current state using the function getTTime (line 8). It puts the event out in the network if it is consistent with the transition (line 17). If the algorithm observes that time k progresses, it checks whether the time k is equal to or greater than the logical time *now* (line 19), if this is the case, it traverses the timed transition and updates the current state (line 19 to 25). In addition, the algorithm traverses transitions that are either internal actions

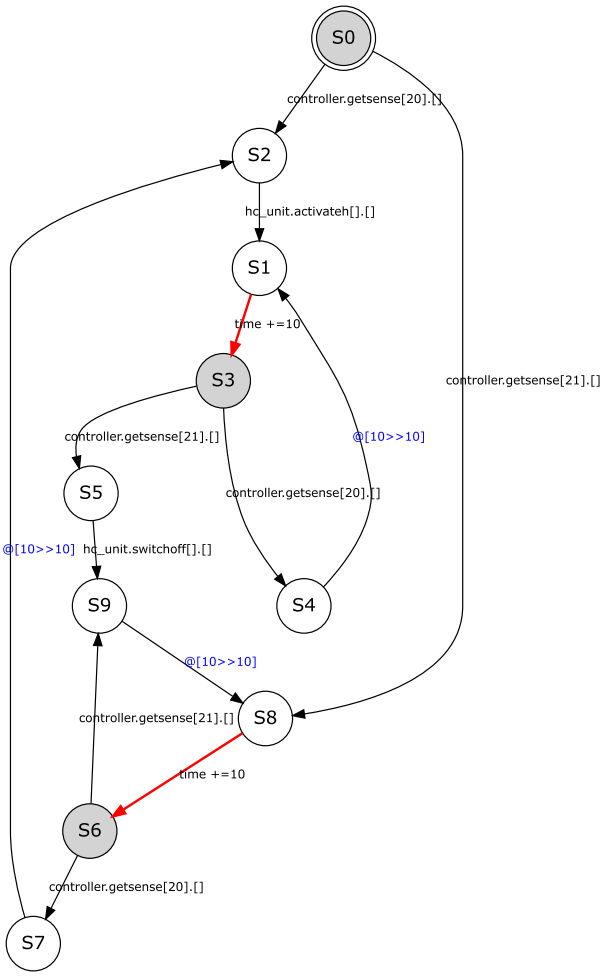


Fig. 7. The Tiny Twin of the transition system of the simplified temperature control system shown in Fig. 5(b).

(*tau*) using function getTTau or time shifting using function getTTime (lines 27 and 44).

Running Example: Monitoring using Tiny Twin. Let the Tiny Twin of Fig. 7 be the input model of the monitor. It sets the current state to S0. The monitor observes the sensor data 20 at time k, i.e. $k = 0$ and compares the sensor data with the label of the outgoing transition. As the sensor data and the label on the outgoing transition at state S0 are the same, the monitor traverses the transition and sets the current state to S2.

The sensor data is then sent out over the network to the controller. The monitor proceeds by observing the control command activateh that is issued by the controller. It traverses the outgoing transition room.activateh since it matches to the command. At state S1, the monitor waits to observe the logical time k advances. If the monitor observes a new sensor data 21 at time $k = 10$, it traverses the timed transition and sets the current state to S3. The monitor compares the sensor data with the label on the outgoing transition at state S3 and sets the current state to S5. The monitor repeats the same process by observing the sensor data or control commands. The monitor makes an alarm and terminates the monitoring process if it observes a sensor data or a control command inconsistent with the model. In the developed module for the *monitor*, it returns an alarm containing (k, e) where k is a time showing at which time during system execution an inconsistency is identified and e is the inconsistent sensor data or control commands at state Si in the Tiny Twin where the *monitor* terminated.

6. Case study: a temperature control system

We evaluate the applicability of our method in detecting and preventing cyber-attacks using a temperature control system case study. This case study is a more complex version of the running example. 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 attackers can alter/inject false sensor data or compromise the controller to tamper with the commands issued by the controller. We developed the Timed Rebeca model of the temperature control system and used Afra to generate the state space. We develop a *monitor* in LF as a reactor to observe and check the system behavior at runtime. The Timed Rebeca model, the LF code of the system and the *monitor* are available on GitHub.[3]

Tiny Twin. The Tiny Twin is created by providing the *ltsconvert* tool with an LTS and the respective silent transitions. The CRTTS of the Timed Rebeca model which is generated by Afra has 799 states and 1440 transitions. The CRTTS is mapped to an LTS with 994 states and 1634 transitions using the *ltscast* function. In the next step, the Tiny Twin is created that has 125 states and 154 transitions.

We show a subset of the state transitions of the Tiny Twin to explain the system behavior at different states (see Fig. 8). In the Tiny Twin of the temperature control system, we see branching states (e.g., $S32$ and $S22$) that present different control flow paths, where the controller decides to activate/switch off the hc_unit regarding the received sensor data. Also, there are some cycles of sensor data transmission and control commands, where the same sensor data and control commands are repeated (e.g., from state S28 to state S20, state S15 to state S122 and state S29 to state S28).

Attack Types and Detection Capability. We consider the number of possible false sensor data and faulty control commands as the number of attacks. In this case, the possible sensor data are considered as (20, 22, 23, or 24) and the possible control commands are (activateh, activatec or switchoff). The combinations of these inputs generate 960 attacks during 80 seconds which is a predefined system execution period. The number of possible false sensors is represented in an abstract way, like in testing we consider different ranges and boundary points. These attacks consist of false sensor data injection attacks that are combined with tampering control commands. In the following, we illustrate the attacks and detection capability of the *monitor* on a run of the system where the actions are shown on the state transitions of the Tiny Twin in Fig. 8. The current temperature of the room is below the desired range, therefore the heating process has been activated (state S32 to state S28). The controller waits for new sensor data and switches off the hc_unit when the temperature goes to the desired range (state S27 to state S68). If the temperature drops below the desired range, the heating process is activated (state S35 to state S30).

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* can detect/drop the faulty control command. From states S32, S21, S20, S35 and S16 you may move to different states. For instance, assume that 21 is sensed as the temperature value in S32 but the *compromised sensor* sends the value 24. According to the Tiny Twin of the case study, the value for the next states can be either 20 (S90), 21 (S22), or 22 (S97) so the *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. So, 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, the *monitor* can not detect the false sen-

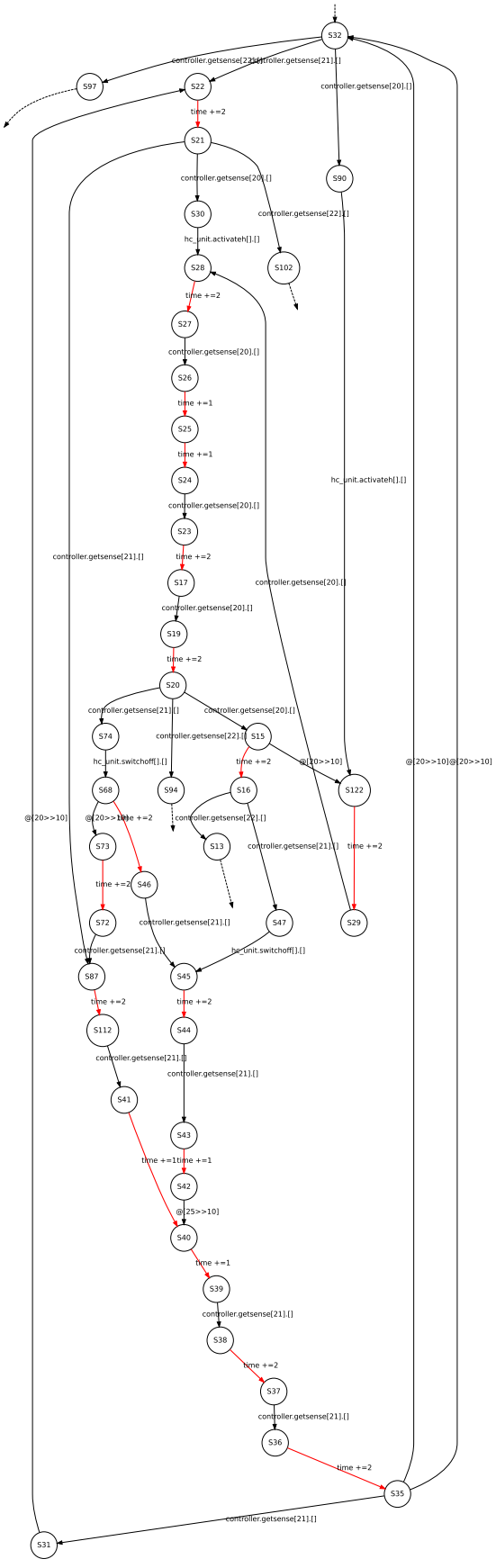


Fig. 8. A subset of the state transitions in the Tiny Twin of the temperature control system. It shows timed transitions, time shifting and branching states.

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

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

System States	*# Attacks*	*False sensor data/ Faulty control commands*	*Detection Capability* (DS/DC)
S32, S21, S20, S35	4	Sensor data (20, 22, 23, or 24)	DS (23 and 24) where 21 is actual sensed value
S27, S24, S17	4	Sensor data (21, 22, 23 or 24)	DS (21, 22, 23 and 24) where 20 is actual sensed value
S46, S44, S112, S39, S37	4	Sensor data (20, 22, 23 or 24)	DS (20, 22, 23 and 24) where 21 is actual sensed value
S90 and S30	2	Command (activatec or switchoff)	DC (activatec and switchoff)
S74	2	Command (activateh or activatec)	DC (activateh and activatec)
S16	4	Sensor data (20, 22, 23 or 24)	DS (20, 23 and 24) where 21 is actual sensed value

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

Table 2
Alarms of the *monitor* in case of attacks.

System States	*False sensor data/ Faulty control commands*	*Alarms list*
S32	Sensor data (23)	[time, y^i : 23, temp : 21]
S27	Sensor data (20)	[time, y^i : 21, temp : 20]
S46	Sensor data (22)	[time, y^i : 22, temp : 21]
G90	Command (activatec)	[time, u^i : activatec, temp : 21]
G74	Command (switchoff)	[time, u^i : switchoff, temp : 21]
S16	Sensor data (24)	[time, y^i : 24, temp : 21]

time: the logical time which is derived using Lingua Franca code, y^i: the inconsistent sensor data, u^i: the inconsistent control command, temp: the stored temperature value in the controller

sor 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 processes stay turned on for a time longer than expected.

Table 2 shows the alarms list returned by the monitor 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 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. 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 LF codes of two case studies (Secure Water Treatment system (SWaT) and Pneumatic Control System (PCS)), for which the *monitor* can properly detect attacks on the system [30]. In the PCS, system dynamics can be affected by environmental factors such as changes in the quality of the air supply. By assigning nondeterministic values to the state variable representing the motion rate of the cylinders in the Timed Rebeca model, we capture the variability of the environment. In the SWaT case study, the process of increasing and decreasing water level is a continuous behavior. We discretize the water level into low, medium, and high categories using state variables. The PCS and SWaT systems are distributed control systems whereas TCS is a centralized control system. 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. 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.

7. Related work

There is a rich literature on using formal models to detect and prevent cyber-attacks on CPS. For instance, authors in [21,6] define the behavior of the system using an automaton and employ it to detect attacks. Authors in [20] model the system as finite state machines and verify the system behavior at runtime. Lanotte et al. [18,19] and Pinisetty et al. [31] propose formal approaches based on runtime enforcement to ensure specification compliance in the control systems. The advantage of our approach is that the model used within the monitor (the Tiny Twin) is not purely a specification. The model is executed (similar to a program) which enables us to debug it, make necessary revisions, and reflect important details back into the model [39].

In [32], Rocchetto and Tippenhauer present a taxonomy of the diverse attacker models for CPSs security and investigate the impact of single-point cyber attacks on SWaT [1]. They [33] use the ASLan++ tool for modeling the physical layer interactions and the CL-AtSe tool for analyzing the state space. Hailesellasie and Hasan [11] verify the PLC network within an industrial control system by creating graphs of the potentially compromised PLC program and a trusted version of the program. The methods in [33,11] prioritize aspects like system architecture or data flow to discover vulnerabilities. They may not explicitly represent the entities and their respective roles. In our approach, we reduce the semantic gap between the model and the entities in real-world applications, enhancing the relevance of the system design because of incorporating actors.

Adepu and Mathur [2] propose a method that detects attacks by identifying anomalies in the behavior of the physical process in the plant. Orpheus [7] monitors the behavior of a device control program based on the invoked system calls, and McLaughlin [25] presents a monitor for the use of electromechanical devices. Compared to the methods that implement monitoring codes to detect attacks (e.g., [2,7,25]), the Tiny Twin does not need coding and embedding within the controllers. The monitor embeds the Tiny Twin and is placed as a proxy within the network where it can handle large plants containing several different sensors and actuators.

Russo et al. [34] propose a lightweight Digital Twin Framework (DTF) to support the quick build of reliable Digital Twins (DTs) for experimental and testing purposes. In [15], authors design a DTF for staging security training and research activities in a replica of a smart grid. The framework includes a module for the automatic execution of attack scripts. Eckhart et al. [9] present CPS Twinning, a DTF aiming at mirroring CPS. Their proposal is inspired by MiniCPS [4], i.e., a framework for real-time CPS simulation. In this study, our focus lies on formal methods for modeling Digital Twins, while other research on Digital Twins (e.g., [34,15,9]) try to design a framework to generate the virtual environment from the specification for testing, monitoring, and security analysis.

Giraldo et al. [10] address various security aspects in CPS applications and assess metrics like false alerts and attack detection probability. They provide an overview of detection mechanisms and their efficacy. Bartocci et al. [5] investigate methods for specifying detection targets, measurement techniques, and system instrumentation.

8. Conclusion

In this paper, we used a Tiny Twin to detect the attacks on sensors and controllers. We employ the mCRL2 *ltsconvert* tool to build the Tiny Twin, which is an abstract version of a state transition system representing the system's correct behavior in the absence of an attack. In our method, we develop a *ltscast* function to map the state space generated by Afra to the LTS that is an input model for the mCRL2. We implement a monitor 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. ltscast function algorithm

Algorithm 2 shows the high-level pseudo-code of *ltscast* function. In the following, we describe the details of the algorithm. The algorithm gets (S, s_0, Act, T) as the input CRTTS and returns (S', s'_0, Act', T') as the LTS. In the input CRTTS, each transition contains a source state, a target state, an action which is a pair of name of the action and the time shifting value. The algorithm performs the mapping process using two loops (lines 6 and 14). In the first loop (lines 6 and 13), we check all transitions if a transition contains a non-zero time shifting value. If a transition is found with a time shifting (line 8), it creates a new state that is the same as the target state and puts the state in S' (line 9). It also divides the transition into two subsequent transitions and puts the transitions in T' (line 10). When a transition is divided, one subsequent transition represents an action, and the other transition represents a time shifting (line 10). If no time shifting is found, it keeps the states in S' and the transition in T' without any change (line 11).

Algorithm 2: ltscast Function.

```
   Input: a CRTTS (S : stateSet, s_0 : initialState, Act : actionSet, T : transitionSet)
   Output: an LTS (S' : stateSet, s'_0 : initialState, Act' : actionSet, T' : transitionSet)
1  begin
2      s'_0 = s_0;
3      stateSet S' = ∅;
4      transitionSet T' = ∅;
5      actionSet Act' = ∅;
6      for each t ∈ T do
7          if t.action.α ≠ 0 then
8              state st = new state();
9              S' = S' ∪ {t.source, t.target, st};
10             T' = T' ∪ {new transition(t.source, t.action.act, st),
                   new transition(st, t.action.α, t.target)};
11         else
12             S' = S' ∪ {t.source, t.target};
13             T' = T' ∪ {new transition(t.source, t.action.act, t.target)};
14     for each t ∈ T' do
15         for each t_1 ∈ T' do
16             if t_1.source = t.source ∧ t_1.action = t.action ∧ t_1.target ≠ t.target then
17                 t_1.action = t_1.action + getDif(t.target, t_1.target);
18                 Act' = Act' ∪ {t_1.action};
19         Act' = Act' ∪ {t.action};
20     return (S', Act', T')
```

In the second loop (line 14 to 19), we find the outgoing transitions of a state which have the same label. If there are two outgoing transitions with the same label from the same source state to different target states (line 14), we tag the transitions using function getDif. The function getDif is used to compare the state variables of the two states (line 17). If the value of a state variable is different in the two states, the function returns the value of the variable in the target state, if the values are the same it returns a zero. The transition is tagged using this value. If there are more than one variable with different values in the two states, the tag is the concatenation of all the values of the variables in the target state. Finally, the labels of the actions are added to the set Act'.

References

[1] S. Adepu, A. Mathur, An investigation into the response of a water treatment system to cyber attacks, in: 2016 IEEE 17th International Symposium on High Assurance Systems Engineering, HASE, IEEE, 2016, pp. 141–148.

[2] S. Adepu, A. Mathur, Distributed attack detection in a water treatment plant: method and case study, IEEE Trans. Dependable Secure Comput. 18 (2018) 86–99.

[3] Afra, An integrated environment for modeling and verifying Rebeca family designs, Online. URL https://rebeca-lang.org/alltools/Afra, 2022. (Accessed 9 December 2022).

[4] D. Antonioli, N.O. Tippenhauer, Minicps: a toolkit for security research on cps networks, in: Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or Privacy, 2015, pp. 91–100.

[5] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, S. Sankaranarayanan, Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications, in: Lectures on Runtime Verification: Introductory and Advanced Topics, 2018, pp. 135–175.

[6] L.K. Carvalho, Y.-C. Wu, R. Kwong, S. Lafortune, Detection and mitigation of classes of attacks in supervisory control systems, Automatica 97 (2018) 121–133.

[7] L. Cheng, K. Tian, D. Yao, Orpheus: enforcing cyber-physical execution semantics to defend against data-oriented attacks, in: Proceedings of the 33rd Annual Computer Security Applications Conference, 2017, pp. 315–326.

[8] J.P. Conti, The day the samba stopped [power blackouts], Eng. Technol. 5 (2010) 46–47.

[9] M. Eckhart, A. Ekelhart, Towards security-aware virtual environments for digital twins, in: Proceedings of the 4th ACM Workshop on Cyber-Physical System Security, 2018, pp. 61–72.

[10] 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 Comput. Surv. 51 (2018) 1–36.

[11] M. Hailesellasie, S.R. Hasan, Intrusion detection in plc-based industrial control systems using formal verification approach in conjunction with graphs, J. Hardw. Syst. Secur. 2 (2018) 1–14.

[12] H. He, J. Yan, Cyber-physical attacks and defences in the smart grid: a survey, IET Cyber-Phys. Syst.: Theory Appl. 1 (2016) 13–27.

[13] C. Hewitt, Viewing control structures as patterns of passing messages, Artif. Intell. 8 (1977) 323–364.

[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] N.K. Kandasamy, S. Venugopalan, T.K. Wong, L.J. Nicholas, Epictwin: an electric power digital twin for cyber security testing, research and education, arXiv preprint, arXiv:2105.04260, 2021.

[16] 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.

[17] E. Khamespanah, M. Sirjani, M. Viswanathan, R. Khosravi, Floating time transition system: more efficient analysis of timed actors, in: Formal Aspects of Component Software, Springer, 2015, pp. 237–255.

[18] R. Lanotte, M. Merro, A. Munteanu, A process calculus approach to detection and mitigation of plc malware, Theor. Comput. Sci. 890 (2021) 125–146.

[19] R. Lanotte, M. Merro, A. Munteanu, Industrial control systems security via runtime enforcement, ACM Trans. Priv. Secur. 26 (2022) 1–41.

[20] E. Lee, Y.-D. Seo, Y.-G. Kim, A cache-based model abstraction and runtime verification for the Internet-of-things applications, IEEE Int. Things J. 7 (2020) 8886–8901.

[21] P.M. Lima, M.V. Alves, L.K. Carvalho, M.V. Moreira, Security against network attacks in supervisory control systems, IFAC-PapersOnLine 50 (2017) 12333–12338.

[22] 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.

[23] 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.

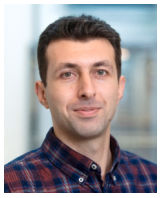
[24] M. Lohstroh, C. Menard, S. Bateni, E.A. Lee, Toward a lingua franca for deterministic concurrent systems, ACM Trans. Embed. Comput. Syst. 20 (2021) 1–27.

[25] S. McLaughlin, CPS: stateful policy enforcement for control system device usage, in: Proceedings of the 29th Annual Computer Security Applications Conference, 2013, pp. 109–118.

[26] R. Mitchell, I.-R. Chen, A survey of intrusion detection techniques for cyber-physical systems, ACM Comput. Surv. 46 (2014) 1–29.

[27] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, M. Caccamo, 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, 2013, pp. 65–74.

[28] 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.
[29] 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.
[30] F. Moradi, S. Abbaspour, B. Pourvatan, Z. Moezkarimi, M. Sirjani, Crystal framework: Cybersecurity assurance for cyber-physical systems (technical report), submitted to NWPT, 2023.
[31] S. Pinisetty, P.S. Roop, S. Smyth, N. Allen, S. Tripakis, R.V. Hanxleden, Runtime enforcement of cyber-physical systems, ACM Trans. Embed. Comput. Syst. 16 (2017) 1–25.
[32] M. Rocchetto, N.O. Tippenhauer, On attacker models and profiles for cyber-physical systems, in: Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Proceedings, Part II 21, Heraklion, Greece, September 26–30, 2016, Springer, 2016, pp. 427–449.
[33] 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.
[34] E. Russo, G. Costa, G. Longo, A. Armando, A. Merlo, Lidite: a full-fledged and featherweight digital twin framework, IEEE Trans. Dependable Secure Comput. (2023).
[35] 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.
[36] M. Sirjani, E. Khamespanah, On time actors, in: Theory and Practice of Formal Methods, Springer, 2016, pp. 373–392.
[37] M. Sirjani, E. Khamespanah, E. Lee, Model checking software in cyberphysical systems, in: COMPSAC 2020, 2020.
[38] M. Sirjani, E.A. Lee, E. Khamespanah, Verification of cyberphysical systems, Mathematics 8 (2020) 1068.
[39] 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, J. Internet Serv. Appl. 12 (2021) 2.
[40] J. Slay, M. Miller, Lessons learned from the maroochy water breach, in: International Conference on Critical Infrastructure Protection, Springer, 2007, pp. 73–82.



Fereidoun Moradi joined the Cyber-Physical Systems Analysis group at Mälardalen University in 2019 as a Ph.D. student. He received his M.Sc. degree in Information Security from the University of Isfahan in 2015, specializing in protocol security analysis. Before starting his Ph.D. studies, Fereidoun worked at Chavoosh Co. (R&D) for 5 years. He recently joined Hitachi Energy as a Senior Cybersecurity Engineer in 2023.

Bahman Pourvatan is a research engineer at Mälardalen University. He works on modeling and analysis of real-time systems using different techniques. Bahman has been a university lecturer for over 25 years, teaching courses on programming, algorithms, and software engineering.

Sara Abbaspour is a lecturer specializing in safety and security-relevant cyber-physical systems. She works in the Cyber-Physical Systems Analysis group at Mälardalen University in Sweden. Sara served as a Postdoctoral researcher from 2018 to 2020 at Mälardalen University. She has successfully completed her PhD and defended her thesis titled 'Concurrency Bugs: Characterization, Debugging, and Runtime Verification'. Her primary research interests encompass safety and security-relevant cyber-physical systems, debugging, testing, and runtime verification of concurrent, parallel, and multicore software, security for wireless networks, service-level agreements in Industrial IoT, autonomous driving, and advanced driver assistance systems (ADAS). Sara also has work experience in various aspects of industrial environments such as Mobile Development Systems, Multimedia Technologies and eLearning applications, RFID, Smart card technologies, and Software System Testing.

Marjan Sirjani is a Professor of Software Engineering at Mälardalen University in Sweden. Marjan has been working on modeling, formal verification, and safety and security assurance of distributed, self-adaptive and cyber-physical systems. Marjan has been the PC member and PC chair of several international conferences including SEFM, FM, FMICS, SAC, and DATE. She is an editor of the journal of Science of Computer Programming. Marjan collaborated with different companies including Volvo CE, Volvo GTO, Volvo Cars, and ABB Robotics. Marjan and her research group designed the actor-based language Rebeca (https://rebeca-lang.org/) in 2001 and are pioneers in building model checking tools, compositional verification theories, and state-space reduction techniques for actors.