A Modest Security Analysis of Cyber-Physical Systems: A Case Study

Ruggero Lanotte[1], Massimo Merro[2], and Andrei Munteanu[2]

[1] Dipartimento di Scienza e Alta Tecnologia, Università dell'Insubria, Como, Italy
ruggero.lanotte@uninsubria.it
[2] Dipartimento di Informatica, Università degli Studi di Verona, Italy
{massimo.merro,andrei.munteanu}@univr.it

Abstract. *Cyber-Physical Systems* (CPSs) are integrations of networking and distributed computing systems with physical processes. Although the range of applications of CPSs include several critical domains, their verification and validation often relies on simulation-test systems rather then formal methodologies. In this paper, we use a recent version of the expressive Modest Toolset to implement a non-trivial engineering application, and test its *safety model checker* prohver as a formal instrument to statically detect a variety of *cyber-physical attacks*, *i.e.*, attacks targeting *sensors* and/or *actuators*, with potential physical consequences. We then compare the effectiveness of the Modest Toolset and its safety model checker in verifying CPS security properties when compared to other state-of-the-art model checkers.

1 Introduction

Cyber-Physical Systems (CPSs) are integrations of networking and distributed computing systems with physical processes, where feedback loops allow the latter to affect the computations of the former and vice versa. CPSs have three main components: the *physical plant*, *i.e.*, the physical process that is managed by the CPS; the *logics*, *i.e.*, controllers, intrusion detection systems (IDSs), and supervisors that govern and control the physical process; the connecting *network*.

Historically, CPSs relied on proprietary technologies and were implemented as stand-alone networks in physically protected locations. However, in recent years the situation has changed considerably: commodity hardware, software and communication technologies are used to enhance the connectivity of these systems and improve their operation.

This evolution has triggered a dramatic increase in the number of attacks to the security of cyber-physical and critical systems, *e.g.*, manipulating sensor readings and, in general, influencing physical processes to bring the system into a state desired by the attacker. Some notorious examples are: (i) the *Stuxnet* worm, which reprogrammed PLCs of nuclear centrifuges in Iran [7], (ii) the attack on a sewage treatment facility in Queensland, Australia, which manipulated the SCADA system to release raw sewage into local rivers [27], or the (iii) the recent cyber-attack on the Ukrainian power grid, again through the SCADA system [15].

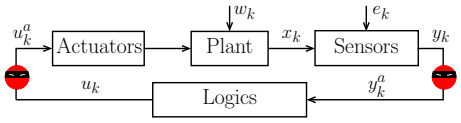

Fig. 1. A threat model for CPSs

The common feature of the systems above is that they are all safety critical and failures may cause catastrophic consequences. Thus, the concern for consequences at the physical level puts *CPS security* apart from standard *IT security*, and demands for *ad hoc* solutions to properly address such novel research challenges.

The *physical plant* of a CPS is often represented by means of a *discrete-time state-space model*[3] consisting of two *difference equations* of the form

$$x_{k+1} = Ax_k + Bu_k + w_k$$
$$y_k = Cx_k + e_k$$

where $x_k \in \mathbb{R}^n$ is the current *(physical) state*, $u_k \in \mathbb{R}^m$ is the *input* (*i.e.*, the control actions implemented through actuators), $w_k \in \mathbb{R}^n$ is the *system uncertainty*, $y_k \in \mathbb{R}^p$ is the *output* (*i.e.*, the measurements from the sensors), and $e_k \in \mathbb{R}^p$ is the *measurement error*. A, B, and C are matrices modelling the dynamics of the physical system.

Cyber-physical attacks typically tamper with both the physical (sensors and actuators) and the cyber layer. In particular, cyber-physical attacks may affect directly the sensor measurements or the controller commands (see Figure 1):

- *Attacks on sensors* consist of reading and possibly replacing the genuine sensor measurements y_k with fake measurements y_k^a.
- *Attacks on actuators* consist of reading, dropping and possibly replacing the genuine controller commands u_k with malicious commands u_k^a, affecting directly the actions the actuators may execute.

One of the central problem in the *safety verification* of CPSs is the *reachability* problem: can an unsafe state be reached by an execution of the system (possibly under attack) starting from a given initial state? In general, the reachability problem for *hybrid systems* (and hence CPSs) is stubbornly undecidable, although boundaries of decidability have been extensively explored in the past couple of decades [1, 14, 17, 30, 26]. Thus, despite the undecidability of the safety problem, a number of formal verification tools for hybrid systems have been recently proposed, based on approximation techniques to obtain an estimation of the set of reachable states: `SpaceEx` [10], `PHAVer` [9] and `SpaceEx AGAR` [3], for linear/affine dynamics, and `HSolver` [23], `C2E2` [6] and `FLOW*` [5], for non-linear dynamics. Among these, the hybrid solver `PHAVer` addresses the *exact verification* of safety properties of hybrid systems with piecewise constant bounds on the derivatives, so-called *rectangular hybrid automata* [14]. *Affine dynamics* are handled by *on-the-fly overapproximation* and partitioning of the state space based on user-provided

[3] See [31] for a taxonomy of the time-scale models used to represent CPSs.

constraints and the dynamics of the system. To force termination and manage the complexity of the computations, methods to conservatively limit the number of bits and constraints are adopted.

Contribution. We implement in the Modest Toolset [12], an integrated collection of tools for the design and the formal analysis of *stochastic hybrid automata*, a simple but totally realistic and nuanced cyber-physical system. The example has been proposed by Lanotte et al. [19] to highlight different classes of attacks on sensors and actuators, in a way that is basically independent on the size of the system. Our case study is implemented in the main modelling language HModest [11], a process-algebra based language that has an expressive programming language-like syntax to design complex systems.

The current version of the toolset comprises several analysis backends, in particular it provides a *safety model checker*, called prohver, that relies on a modified version of the hybrid solver `PHAVer` [9]. We use prohver to analyse three simple but significative cyber-physical attacks targeting sensors and/or actuators of our case study by compromising either the corresponding physical device or the communication network used by the device. The three attacks have already been carefully studied in [19] focussing on the time aspects of the attacks (begin, duration, *etc.*) and the physical impact on the system under attack (deadlock, unsafe behaviour, *etc.*). Here, we test the safety model checker prohver as an automatic tool to get the same (or part of the) results that have been manually proved in [19]. We then compare its effectiveness in verifying CPS security properties, when compared to other state-of-the-art models checkers, such as PRISM [16], UPPAAL [2] and Real-Time Maude [22].

Outline. In Section 2 we give a brief description of the Modest Toolset. In Sections 3 and 4 we first describe and then implement in HModest our case study. In Section 5 we put under stress the safety model checker prohver for a security analysis of our case study under three different cyber-physical attacks. In Section 6 we draw conclusions, compare the expressivity of the Modest Toolset with respect to other model-checkers, and discuss related work in the context of formal methods for CPS security.

2 The Modest Toolset

The Modest Toolset [4] has been originally proposed as an integrated collection of tools for the design and the formal analysis of stochastic timed automata (STA). More recently, it has been extended to add differential equations and inclusions as an expressive way to model continuous system evolutions [11]. Thus, the current version of the toolset [12] is now based on the rich semantic foundation of networks of stochastic hybrid automata (SHA), *i.e.*, sets of automata that run asynchronously and can communicate via shared actions and global variables.

SHA combine three key modelling concepts:

- *Continuous dynamics* to represent continuous processes, such as physical laws or chemical reactions, the evolution of general continuous variables over time can be described using differential (in)equations.
- *Nondeterminism* to model concurrency (via an interleaving semantics) or the absence of knowledge over some choice, to abstract from details, or to represent the influence of an unknown environment.
- *Probability* to model situations in which an outcome is uncertain but the probabilities of the outcomes are known; these choices may be discrete ("probabilistic") or continuous ("stochastic").

The current version of the MODEST TOOLSET comprises analysis backends for model checking timed automata (mctau) and probabilistic timed automata (mcpta), and for statistical model checking of stochastic timed automata (modes). However, in this paper we focus on the *safety model checker* for SHA, called prohver, that relies on a modified version of the hybrid solver `PHAVer` [9].

The main modelling language is HMODEST [11], a process-algebra based language that has an expressive programming language-like syntax to design complex models in a reasonably concise manner. Here, we provide a brief and intuitive explanation of the main constructs.

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

In order to better explain these constructs, we model a small example described by means of a standard timed process-calculus notation (say, Hennessy and Regan's TPL [13]). Consider a *Master* and a *Slave* process that may synchronise via a private synchronisation channel *sync*, and use a private channel *ins* to allow the *Master* to send instructions to the *Slave*. Depending on the received instructions, the *Slave* either synchronises with the *Master* and then restart, or sleeps for one time unit and then ends its execution. Once synchronised, the *Master* sleeps for two time units. Formally,

$$\begin{aligned} \mathit{Master} &\stackrel{\text{def}}{=} \mathit{ins}\langle\mathsf{go}\rangle; \mathit{sync}; \mathsf{sleep}(2); \mathit{ins}\langle\mathsf{end}\rangle \\ \mathit{Slave} &\stackrel{\text{def}}{=} \mathit{ins}(i); \mathsf{if}\ (i = \mathsf{go})\ \{\mathit{sync}; \mathit{Slave}\}\ \mathsf{else}\ \{\mathsf{sleep.stop}\} \end{aligned}$$

and the compound system is given by

$$(\mathit{Master} \parallel \mathit{Slave}) \backslash \{\mathit{ins}, \mathit{sync}\}\,.$$

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

// main behaviour
par { :: Master() :: Slave()
}
```

Fig. 2. Master and Slave processes in HMODEST

Figure 2 shows an implementation in HMODEST of the system above. Both master and slave declare private clocks that are reset each time is necessary to impose a specific time delay. Value-passing communication is implemented via the two actions *go* and *end*; the testing via nondeterministic choice.

Besides these operators, the case study that we will present in the next section includes specifications over continuous variables, such as constraints over the derivate of continuous variables of the form $a \leq \dot{x} \leq b$, with a and b constant (as in *rectangular hybrid automata*), or nondeterministic initialisations of the form $z \in [a, b]$. The former requirement is realised in HMODEST by means of an invariant construct: $invariant(der(x \geq a)$ &&$\ der(x \leq b))$. The latter constraint is implemented via the *any* construct. For instance, $any(z, z >= a$ &&$\ z <= b)$ returns a value nondeterministically chosen in the real interval $[a, b]$.

The safety model checker prohver allows the verification of reachability properties of the form $Pmax(\Diamond_{time \leq T}\ e)$. This query returns un upper bound of the probability of reaching the states characterised by the deterministic expression e within the time bound T.[4] Moreover, as the models may be nondeterministic, $Pmax()$ computes the probability over all possible resolutions of nondeterminism.

3 A case study

In this section, we describe the case study recently introduced in [19]. Here, we wish to remark that while the example is quite simple, it is actually far from trivial and designed to describe a wide number of attacks. Furthermore, for simplicity, in the description of the case study we use a discrete-time model, although in its implementation we will adopt a continuous notion of time.

Consider a CPS *Sys* in which the temperature of an engine is maintained within a certain range by means of a cooling system controlled by a controller.

[4] Later in the paper, we will show how to get the exact probability.

The system is also equipped with a IDS that does runtime safety verification. Let's describe both the physical and the cyber component of the CPS *Sys*.

The physical environment of *Sys* is constituted by:

- a *variable temp*, initialised to 0, for the current temperature of the engine;
- a *sensor sens* measuring the temperature of the engine;
- an *actuator cool* to turn on/off the cooling system; *cool* ranges over the set $\{-1, +1\}$ to denote active and inactive cooling, respectively;
- the *evolution equation* $temp_{k+1} = temp_k + cool_k + w_k$, where $w_k \in [-0.4, +0.4]$ denotes the *uncertainty* associated to *temp*; thus the variable *temp* is increased (*resp.*, is decreased) of one degree per time unit if the cooling system is inactive (*resp.*, active) up to a bounded uncertainty w_k;
- a *measurement equation* $sens_k = temp_k + e_k$, where $e_k \in [-0.1, +0.1]$ denotes the *noise* associated to the sensor *sens*;
- an *invariant function* returning the Boolean true if the state variable *temp* lays in the interval $[0, 20]$, false otherwise;
- a *safety function* returning the Boolean true if the safety conditions are satisfied, false otherwise; the safety of the CPS depends on a (fictitious) variable *stress* keeping track of the level of stress of the mechanical parts of the engine due to high temperatures; *stress* ranges over the set $\{0, 1, 2, 3, 4, 5\}$, where 0 means no stress and 5 high stress; formally, $stress_{k+1} = \min(5, stress_k + 1)$ if $temp_k > 9.9$, while $stress_{k+1} = 0$ if $temp_k \leq 9.9$.

Let us define the cyber component of the CPS *Sys*. For simplicity, we use a simple process-calculus notation similar to that of Lanotte and Merro's `CaIT` [18]. The logics of *Sys* is modelled by means of two parallel processes: *Ctrl* and *IDS*. The former models the *controller* activity, consisting in reading the temperature of the engine and in governing the cooling system; whereas the latter models a simple *intrusion detection system* that attempts to detect and signal abnormal behaviours of the system. Intuitively, *Ctrl* senses the temperature of the engine via the sensor *sens* (reads the sensor) at each time slot. When the *sensed temperature* is above 10 degrees, the controller activates the coolant via the actuator *cool* (sending a command to the actuator). The cooling activity is maintained for 5 consecutive time units. After that time, the controller synchronises with the *IDS* component via a synchronisation channel *sync*, and then waits for *instructions*, via a value-passing channel *ins*. The *IDS* component checks whether the *sensed temperature* is still above 10. If this is the case, it sends an *alarm* of "high temperature", via a specific channel, and then says to *Ctrl* to keep cooling for a further 5 time units; otherwise, if the temperature is not above 10, the *IDS* component requires *Ctrl* to stop the cooling activity. More formally,

$$
\begin{aligned}
\mathit{Ctrl} &= \mathsf{read}\ \mathit{sens}(x).\mathsf{if}\ (x > 10)\ \{\mathit{Cooling}\}\ \mathsf{else}\ \{\mathsf{sleep}.\mathit{Ctrl}\}\\
\mathit{Cooling} &= \mathsf{write}\ \mathit{cool}\langle\mathsf{on}\rangle.\mathsf{sleep}(5).\mathit{Check}\\
\mathit{Check} &= \mathit{sync}.\mathit{ins}(y).\mathsf{if}\ (y = \mathsf{keep\_cooling})\ \{\mathsf{sleep}(5).\mathit{Check}\}\\
&\qquad \mathsf{else}\ \{\mathsf{write}\ \mathit{cool}\langle\mathsf{off}\rangle.\mathsf{sleep}.\mathit{Ctrl}\}\\
\mathit{IDS} &= \mathit{sync}.\mathsf{read}\ \mathit{sens}(x).\mathsf{if}\ (x > 10)\ \mathit{ins}\langle\mathsf{keep\_cooling}\rangle.\{\mathit{alarm}\langle\mathsf{high\_temp}\rangle.\mathsf{sleep}.\mathit{IDS}\}\\
&\qquad \mathsf{else}\ \{\mathit{ins}\langle\mathsf{stop}\rangle.\mathsf{sleep}.\mathit{IDS}\}
\end{aligned}
$$

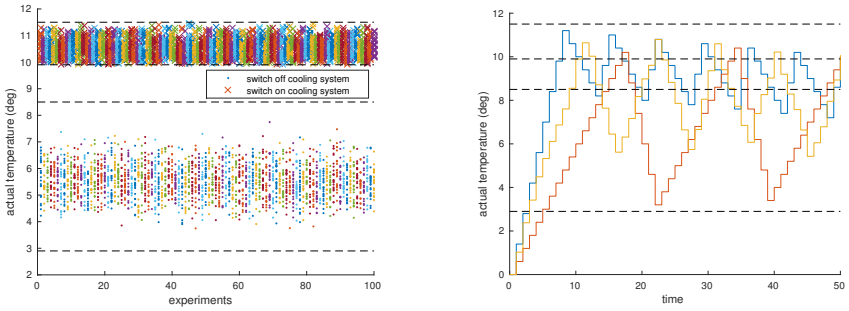

Fig. 3. Simulations in MATLAB of *Sys*

The whole cyber component of *Sys* is given by the parallel composition of the two processes *Ctrl* and *IDS* in which the channels *sync* and *ins* have been restricted: $(Ctrl \parallel IDS)\backslash\{sync, ins\}$.

In Figure 3, the left graphic collect a campaign of 100 simulations of our engine in MATLAB, lasting 250 time units each, showing that the value of the state variable *temp* when the cooling system is turned on (*resp.*, off) lays in the interval $(9.9, 11.5]$ (*resp.*, $(2.9, 8.5]$); these bounds are represented by the dashed horizontal lines. The right graphic of the same figure shows three possible evolutions in time of the state variable *temp*: (i) the first one (in red), in which the temperature of the engine always grows as slow as possible and decreases as fast as possible; (ii) the second one (in blue), in which the temperature always grows as fast as possible and decreases as slow as possible; (iii) and a third one (in yellow), in which, depending whether the cooling is off or on, temperature grows or decreases of an arbitrary offset laying in the interval $[0.6, 1.4]$.

4 An implementation in HModest

In this section, we provide our implementation in HModest of the case study presented in the previous section. The whole system is divided in three high level processes running in parallel (see Figure 4):

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

The process Plant() consists of the parallel composition of four processes: Engine(), Actuators(), Sensors() and Safety() (see Figure 5). The former models the dynamics of the variable *temp* depending on the cooling activity. The temperature evolves in a continuous manner, and its rate is described by means of differential inclusions of the form $a \leq \dot{x} \leq b$ implemented via the construct *invariant*. The *on* action triggers the coolant and drives the process Engine() into a state CoolOn() in which the temperature decreases at a rate comprised in the range

```
// global clock and global action declarations
clock global_clock;
action on, off;
...
// global variable declarations
var sens = 0; der(sens) = 0;
bool safe = true;
bool is_deadlock = false;
...
//process declarations
process Plant() {
  var temp = 0;
  ...
  par { :: Engine() :: Sensors() :: Actuators() :: Safety() }
}
process Logics() {
  ...
  par { :: Ctrl() :: IDS() }
}
process Network() {
  ...
  par { :: Proxy_actuator() :: Proxy_sensor() }
}

// main
par { :: Plant() :: Logics() :: Network() }
```

Fig. 4. Implementation in HMODEST of *Sys*

[−DT−UNCERT, −DT+UNCERT]. On the other hand, in the presence of a *off* action the engine moves into a CoolOff() state in which the coolant is turned off, so that the temperature increases at a rate ranging in [DT−UNCERT, DT+UNCERT].

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

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

The last parallel component of the process Plant() is the process Safety(). This process defines a local variable *stress* depending on the temperature reached by the engine; we recall that $stress = 0$ denotes no stress while $stress = 5$ represents maximum stress. Here, is worth mentioning that the variable *stress* could be implemented either as a bounded integer variable, which would increase the discrete complexity of the underlying hybrid automaton, or as a continuous variable with dynamics set to zero (*i.e.*, $der(stress) = 0$) that would increase the continuous complexity of the automaton. We have adopted the second option as it ensures better performances. The Safety() process sets the global Boolean variable *safe* to false only when the system reaches the maximum stress, *i.e.*, $stress = 5$, and reset it to true otherwise. Thus, this variable says when the CPS is currently in a state that is violating the *safety conditions*. Similarly, the global Boolean variable *is_deadlock* is set to true whenever the system invariant is violated; in that case the whole CPS stops.

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

Fig. 5. Plant() sub-processes

The process Logics() consists of the parallel composition of two processes: Ctrl() and IDS() (see Figure 6). The former senses the temperature by triggering a *read_sensor_ctrl* action to request a measurement and waits for an *ack_sensor_ctrl* action to read the measurement in the variable *sens*. Depending on the value of *sens* the controller decides whether to activate or not the cooling system. If $sens \leq 10$ the process sleeps for one time unit and then check the temperature again. If $sens > 10$ then the controller activates the coolant by emitting the *set_cool_on* action that will reach the Engine() (via the Network()'s proxy). Afterwards the control passes to the process Check() that verifies whether the current cooling activity is effective in dropping the temperature below 10. The process Check() maintains the cooling activity for 5 consecutive time units. After that, it synchronises with the process IDS() via the action *sync_ids*, and waits

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

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

Fig. 6. Logics() sub-processes

for instructions from IDS(): (i) *keep cooling* for other 5 time units and then check again, or (ii) *stop the cooling* activity and returns. These two instructions are represented by means of the actions *keep_cooling* and *stop_cooling*, respectively.

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

The process Network() consists of the parallel composition of two processes: Proxy_actuator() and Proxy_sensor() (see Figure 7). The former provides the remote actuation. Basically, it forwards the actuators commands originating from the process Ctrl() to the process Actuators(). The process Proxy_sensor() waits for requests of measurement originating from processes Ctrl() or IDS() (we use different actions for each of them) and relay these requests to the process Sensor()

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

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

Fig. 7. Network() process

that implements the measurement equation. When the temperature has been detected an ack signal is returned and propagated up to the requesting process.

Verification. We conduct our safety verification using 4 notebooks with the following set-up: (i) 2.8 GHz Intel i7 7700 HQ, with 16 GB memory, and Linux Ubuntu 16.04 operating system; (ii) Modest Toolset Build 3.0.23 (2018-01-19).

In order to assess the correct functioning of our implementation, we verify a number of properties of our CPS *Sys* by means of the safety model checker prohver. Here, it is important to recall that prohver relies on the hybrid solver `PHAVer` which computes an *overapproximation* of the reachable states to ensure termination and accelerate convergence [9]. As consequence, the probability returned by the verification of a generic property $Pmax(\Diamond_T e_{prop})$ is an upper bound of the exact probability, and hence it is significant only when equal to zero (*i.e.*, when the property is not satisfied). However, as our CPS *Sys* presents a *linear dynamics* it is possible to compute the exact probability by launching our analyses with the no_cheap_contain_return_others flag (see [8]) which enables the exact computation of the reachable sets, with obvious implications on the time required to complete the analyses. As our case study does not present a probabilistic behaviour, the results of our analyses will always range in the set $\{0, 1\}$ (unsatisfied/satisfied) with a 100% accuracy.

Furthermore, as a formula $\Box e$ is satisfied if and only if $\Diamond \neg e$ is unsatisfied, we can use prohver to verify properties expressed in terms of *time bounded LTL* formulae of the form $\Box_{[0,T]} e_{prop}$ or $\Diamond_{[0,T]} e_{prop}$. Actually, in our analyses we will always verify properties of the form $\Box_{[0,T]} e_{prop}$, relying on the quicker

overapproximation when proving that the property is satisfied, and resorting to the slower exact computation when proving that the property is not satisfied.

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

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

More generally, our implementation of *Sys* satisfies the following three properties:
- $\Box_{[0,100]}(\neg deadlock)$, saying that the system does not deadlock;
- $\Box_{[0,100]}(safe)$, saying that the system does not violate the safety conditions;
- $\Box_{[0,100]}(\neg alarm)$: saying that the IDS does not fire any alarm.

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

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

5 A Static Security Analysis

In this section, we use the safety model checker prohver to test its limits when doing static security analysis of CPSs. In particular, we implement three simple cyber-physical attacks targeting our system *Sys*:

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

These attacks are implemented by tampering with either the physical devices (actuators and/or sensors) or the communication network (*man-in-the-middle*). In order to implement an attack on the sensor (*resp.*, actuator) we suppose the attacker is able to compromise the Sensors() (*resp.*, Actuators()) process. Whereas the attacks targeting the communication network compromise either the Proxy_sensor() or the Proxy_actuator() process, depending whether they are targeting the sensor or the actuator. In general, attacks on the communication network do not require a deep knowledge on the physical dynamics of the CPS.

Attack 1. The first attack targets the actuator *cool* in a very simple manner. It operates exclusively in a specific time instant m, when it tries to drop the command to turn on the cooling system coming from the controller. Figure 8 shows the implementation of this man-in-the-middle attack compromising the Proxy_actuator() process.

We recall that the controller will turn on the cooling system only if it senses a temperature above 10 (as NOISE = 0.1, this means $temp > 9.9$). It is not

```
process E_Proxy_actuator(){
   clock c;
      do{ alt{ :: set_cool_on {= c = 0=};
                  invariant(c <= 0) when(c >= 0)
                  alt{ // drop the cool_on command in the time instant m
                       :: when(global_clock == m) tau
                       // in the other time instants forward correctly
                       :: when(global_clock < m || global_clock > m) cool_on_actuator
                  }
               :: set_cool_off {= c = 0=};
                  invariant(c <= 0) when(c >= 0) cool_off_actuator
          }
       }
}
```

Fig. 8. DoS attack to the actuator

difficult to see that this may happen only if $m > 7$ (in the time instant 7 the maximum temperature that may be reached by the engine is $7 \cdot (\text{DT+UNCERT}) = 7 \cdot (1 + 0.4) = 9.8$ degrees). Since the process Ctrl() never re-send commands to the actuator, if the attacker is successful in dropping the command to turn on the cooling system in the time slot m then the temperature will continue to rise, and after 2 time instants, in the time instant $m + 2$, the system will violate the safety conditions. This is noticed by the IDS() that will fire alarms every 5 time instants, until the CPS deadlocks because $temp > 20$.

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

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

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

Attack 2. The second attack compromises the sensor in order to provide fake measurements to the controller. The compromised sensor operates as follows: (i) in any time instant smaller than or equal to 1 the sensor works correctly, (ii) in any time instant greater than 1 the sensor returns the temperature sensed at time 1. Figure 9 provides an implementation of the compromised sensor.

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

Fig. 9. DoS attack to the sensor

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

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

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

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

Attack 3. Our last attack is a variant of the previous one as it provides the controller with a temperature decreased by an offset (in this case 2), for n consecutive time instants. Unlike the previous attack, in case of encrypted communication, this attack cannot be mounted in the network as it requires the knowledge of the measurement equation. Figure 10 shows the implementation of a compromised sensor device acting as required. Basically, when *global_clock* $<= n$ the compromised sensor returns a measurement affected by the offset; on the

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

Fig. 10. Integrity attack to the sensor device

other hand, when $global\_clock > n$ the sensor works correctly and returns the authentic measurement.

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

- For $n \leq 7$ the attack is harmless as the variable *temp* may not reach a (critical) temperature above 9.9; thus, all properties seen for the system in isolation remain valid when the system is under attack.
- For $n = 8$, the variable *temp* might reach a temperature above 9.9 and the attack would delay the activation of the cooling system of one time instant. As a consequence, the system might get into an unsafe state in the time instants 12 and 13, but no alarm will be fired (*stealthy attack*). This is proved by verifying the following properties:
  - $\Box_{[0,100]}((global\_clock < 12 \ \vee \ global\_clock > 14) \implies safe)$ ✔
  - $\Box_{[0,100]}((global\_clock \leq 12 \ \wedge \ global\_clock \geq 12) \implies safe)$ ✖
  - $\Box_{[0,100]}((global\_clock \leq 13 \ \wedge \ global\_clock \geq 13) \implies safe)$ ✖
  - $\Box_{[0,100]}(\neg alarm)$ ✔.
- For $n > 8$ the system may get into an unsafe state in a time instant between 12 and $n+12$. The IDS will fire the alarm but it will definitely miss a number of violations of safety conditions as after the instant $n+6$ it does not fire any alarm, although we prove there are unsafe states. This is a *temporary attack* as the system behaves correctly after the time instant $n+12$. Summarising:
  - $\Box_{[0,100]}(\neg deadlock)$ ✔
  - $\Box_{[0,100]}((global\_clock < 12 \ \vee \ global\_clock > n+12) \implies safe)$ ✔
  - $\Box_{[0,100]}((global\_clock \geq 12 \ \wedge \ global\_clock \leq n+12) \implies safe)$ ✖
  - $\Box_{[0,100]}((global\_clock > n+6 \ \wedge \ global\_clock \leq n+12) \implies safe)$ ✖
  - $\Box_{[0,100]}((global\_clock < n{+}1 \ \vee \ global\_clock > n+6) \implies \neg alarm)$ ✔
  - $\Box_{[0,100]}((global\_clock \geq n{+}1 \ \wedge \ global\_clock \leq n+6) \implies \neg alarm)$ ✖.

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

6 Conclusions

As said in the Introduction, the safety model checker within the MODEST TOOLSET relies on a modified version of the hybrid solver `PHAVer`, whose specification language is a slight variation of hybrid automata supporting compositional reasonings, where input and output variables are clearly distinguished [20]. Although, `PHAVer` would be a good candidate for the verification of small CPSs, we preferred to specify our case study in the high-level language HMODEST, supporting: (i) differential inclusion to model linear CPSs with constant bounded derivatives; (ii) linear formulae to express nondeterministic assignments within a dense interval; (iii) compositional programming style inherited from process algebra (*e.g.*, parallel composition, nondeterministic choice, loops, *etc.*); (iv) shared actions to synchronise parallel components.

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

For the security analysis we have used the safety model checker prohver. Basically, we have verified LTL properties on the system under attack. Although, we have verified most of the properties that have been manually proved in [19], we have not been able to capture time properties on the responsiveness of the IDS to violations of the safety conditions. Properties such as:

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

Note that prohver has been designed to do *probabilistic model-checking*, while in this paper we only do model checking. Actually, one of the reasons why we implemented our case study in HMODEST is because we aim at strengthening our security analysis by resorting to probabilistic model checking. This would allow us to replace nondeterministic uncertainty and nodeterministic noise with probability distributions (for instance, *normal distributions* are very common in this context).

A comparison with other model-checkers. We tried to verify our case study also with other model-checkers for distributed systems providing high-level specification languages and expressive query languages, such as PRISM [16], UPPAAL [2] and Real-Time Maude [22]. In particular, as our example has a discrete notion of time we started looking at verification tools supporting discrete time.

PRISM, for instance, relies on Markov decision processes or discrete-time Markov chains, depending whether one is interested in modelling nondeterminism or not. It supports the verification of both CTL and LTL properties (when dealing with nonprobabilistic systems). This allowed us to express the formula $\exists\Diamond(\Box_{[t,t+5]} temp > 9.9)$ to verify violations of the safety conditions, avoiding the implementation of the Safety() process. However, using integer variables to represent state variables with a fixed precision requires the introduction of extra transitions (to deal with nondeterministic errors) that significantly complicates the PRISM model.

In this respect, UPPAAL appears to be more efficient than PRISM, as we have been able to concisely express the error occurring in integer state variables thanks to the *select()* construct, in which the user can fix the granularity adopted to approximate a dense interval. This discrete representation provides an *under-approximation* of the system behaviour; thus, a finer granularity translates into an exponential increase of the complexity of the system, with obvious consequences on the verification performance. UPPAAL has provided us with a simple way to implement the preemptive power of cyber-physical attacks by assigning priorities to processes. Thus, a system under attack can be easily represented by simply putting in parallel the system and the attacker. The tool supports the verification of a simplified version of CTL properties (no nesting of path formulae is allowed). Thus, as in HMODEST, we cannot express the formula $\exists\Diamond(\Box_{[t,t+5]} temp > 9.9)$ and we had to implement a Safety() process.

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

Formal methods for CPS security. A few works use formal methods for CPS security, although they apply methods, and most of the time have goals, that are quite different from ours. As already said, the case study has been taken from [19]. In that paper the authors present a threat model for a formal study of a variety of cyber-physical attacks. They also propose a formal technique to assess the tolerance of CPSs to classes of attacks. The paper provides a stepping stone for formal and automated analysis techniques for checking the security of CPSs.

In [28, 29], Vigo presents an attack scenario that addresses some of the peculiarities of a cyber-physical adversary, and discussed how this scenario relates to other attack models popular in the security protocol literature. Unlike us, this paper focuses on DoS attacks without taking into consideration timing aspects. Rocchetto and Tippenhaur [25] introduce a taxonomy of the diverse attacker models proposed for CPS security and outline requirements for generalised attacker models; in [24], they then propose an extended Dolev-Yao attacker model suitable for CPS security. In their approach, physical layer interactions are modelled as abstract interactions between logical components to support reasoning on the physical-layer security of CPSs. This is done by introducing additional orthogonal channels. Time is not represented. Nigam et al. [21] work around the notion of Timed Dolev-Yao Intruder Models for Cyber-Physical Security Protocols by bounding the number of intruders required for the automated verification of such protocols. Following a tradition in security protocol analysis, they provide an answer to the question: How many intruders are enough for verification and where should they be placed? They also extend the strand space model to CPS protocols by allowing for the symbolic representation of time, so that they can use Real-Time Maude [22] along with SMT support. Their notion of time is however different from ours, as they focus on the time a message needs to travel from an agent to another. The paper does not mention physical devices, such as sensors and/or actuators.

Acknowledgements. We thank the anonymous reviewers for their insightful and careful reviews that allowed us to significantly improve the paper. We thank Fabio Mogavero for stimulating discussions on model checking tools, and Arnd Hartmanns for "tips and tricks" on the Modest Toolset. This work has been partially supported by the project "Dipartimenti di Eccellenza 2018-2022" funded by the Italian Ministry of Education, Universities and Research (MIUR).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183 – 235 (1994)
2. Behrmann, G., David, A., G., L.K., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: D'Argenio, P., Miner, A., Rubino, G. (eds.) QEST 2006. pp. 125–126. IEEE Computer Society (2006). https://doi.org/10.1109/QEST.2006.59
3. Bogomolov, S., Frehse, G., Greitschus, M., Grosu, R., S., P.C., Podelski, A., Strump, T.: Assume-Guarantee Abstraction Refinement Meets Hybrid Systems. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 116–131. Springer (2014). https://doi.org/10.1007/978-3-319-13338-6
4. Bohnenkamp, H.C., Hermanns, H., Katoen, J.P.: motor: The modestTool environment. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 500–504. Springer (2007). https://doi.org/10.1007/978-3-540-71209-1
5. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An Analyzer for Non-linear Hybrid Systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8

6. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: A Verification Tool for Stateflow Models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0
7. Falliere, N., Murchu, L., Chien, E.: W32.Stuxnet Dossier (2011)
8. Frehse, G.: Phaver language overview v0.35 (2006), `http://www-verimag.imag.fr/~frehse/phaver_web/phaver_lang.pdf`
9. Frehse, G.: Phaver: Algorithmic verification of hybrid systems past hytech. International Journal on Software Tools for Technology Transfer **10**(3), 263–279 (2008)
10. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable Verification of Hybrid Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1
11. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.: A compositional modelling and analysis framework for stochastic hybrid systems. Formal Methods in System Design **43**(2), 191–232 (2013)
12. Hartmanns, A., Hermanns, H.: The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8
13. Hennessy, M., Regan, T.: A process algebra for timed systems. Information and Computation **117**(2), 221–239 (1995)
14. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? Journal of Computer and System Sciences **57**(1), 94 – 124 (1998)
15. ICS-CERT: Cyber-Attack Against Ukrainian Critical Infrastructure, https://ics-cert.us- cert.gov/alerts/IR-ALERT-H-16-056-01
16. Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1
17. Lafferriere, G., Pappas, G.J., Sastry, S.: O-minimal hybrid systems. Mathematics of Control, Signals, and Systems **13**(1), 1–21 (2000)
18. Lanotte, R., Merro, M.: A semantic theory of the Internet of Things. Information and Computation **259**(1), 72–101 (2018)
19. Lanotte, R., Merro, M., Muradore, R., Viganò, L.: A Formal Approach to Cyber-Physical Attacks. In: CSF 2017. pp. 436–450. IEEE Computer Society (2017). https://doi.org/10.1109/CSF.2017.12
20. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid I/O automata. Information and Computation **185**(1), 105–157 (2003)
21. Nigam, V., Talcott, C., Urquiza, A.A.: Towards the Automated Verification of Cyber-Physical Security Protocols: Bounding the Number of Timed Intruders. In: Askoxylakis, I.G., Ioannidis, S., Katsikas, S.K., Meadows, C.A. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 450–470. Springer (2016). https://doi.org/10.1007/978-3-319-45741-3
22. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2), 161–196 (2007)
23. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. ACM Transactions on Embedded Computing Systems **6**(1), 8 (2007)
24. Rocchetto, M., Tippenhauer, N.O.: CPDY: Extending the Dolev-Yao Attacker with Physical-Layer Interactions. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 175–192 (2016). https://doi.org/10.1007/978-3-319-47846-3

25. Rocchetto, M., Tippenhauer, N.O.: On Attacker Models and Profiles for Cyber-Physical Systems. In: Askoxylakis, I.G., Ioannidis, S., Katsikas, S.K., Meadows, C.A. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 427–449. Springer (2016). https://doi.org/10.1007/978-3-319-45741-3
26. Roohi, N.: Remedies for building reliable cyber-physical systems. Ph.D. thesis, University of Illinois at Urbana-Champaign (2017)
27. Slay, J., Miller, M.: Lessons Learned from the Maroochy Water Breach. In: Goetz, E., Shenoi, S. (eds.) ICCIP 2007. IFIP, vol. 253, pp. 73–82. Springer (2007). https://doi.org/10.1007/978-0-387-75462-8_6
28. Vigo, R.: The Cyber-Physical Attacker. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7613, pp. 347–356. Springer (2012). https://doi.org/10.1007/978-3-642-33675-1
29. Vigo, R.: Availability by Design: A Complementary Approach to Denial-of-Service. Ph.D. thesis, Danish Technical University (2015)
30. Vladimerou, V., Prabhakar, P., Viswanathan, M., Dullerud, G.: STORMED hybrid systems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 136–147. Springer (2008). https://doi.org/10.1007/978-3-540-70583-3
31. Zacchia Lun, Y., D'Innocenzo, A., Malavolta, I., Di Benedetto, M.D.: Cyber-Physical Systems Security: a Systematic Mapping Study. CoRR **abs/1605.09641** (2016), `http://arxiv.org/abs/1605.09641`