Journal of Computer Virology and Hacking Techniques manuscript No.
(will be inserted by the editor)

Formalization and Co-simulation of Attacks on Cyber-physical Systems

Cinzia Bernardeschi · Andrea Domenici · Maurizio Palmieri

Abstract This paper presents a methodology for the formal modeling of security attacks on cyber-physical systems, and the analysis of their effects on the system using logic theories. We consider attacks only on sensors and actuators. A simulated attack can be triggered internally by the simulation algorithm or interactively by the user, and the effect of the attack is a set of assignments to the variables defined in the Controller. The global effects of the attacks are studied by injecting attacks in the system model and co-simulating the overall system, including the system dynamics and the control part. Interesting properties of the behaviour of the system under attack can also be formally proved by theorem proving. The INTO-CPS framework has been used for co-simulation, and the methodology is applied to the Line follower robot case study of the INTO-CPS project. The theorem prover of PVS has been used for deriving formal proofs of invariants of the system under attack.

Keywords Security, Cyber-physical attacks, Co-simulation, Formal verification

1 Introduction

Model-based design of cyber-physical systems (CPS) allows us to analyze the system behavior before a physical prototype of the system is built. Simulation is one of the techniques that are usually applied together with testing in the analysis of systems behaviors. In the case of cyber-physical systems, simulation often takes place in the form of co-simulation, which allows sub-systems, each modeled with its most appropriate languages and tools, to be composed together. The main advantage of co-simulation is modeling flexibility, because it does not require a single modeling language for all system parts (e.g., discrete and continuous parts). The Functional Mockup Interface (FMI) [8] is an emerging standard for co-simulation of cyber-physical systems.

Moreover, model-based design based on formal methods enables proofs of correctness for the system. Formal methods have been used intensively in the past in the development of safety-critical systems, and they are also assuming a fundamental role in the security field. The main advantage of formal methods in the field of security is that they are the only technique that can be used to formally prove resilience to attacks. For example, [24] reports on the history of application of formal methods to cryptographic protocol analysis, in [3] abstract interpretation was applied to certify programs for secure information flow, and in [25] model checking is applied in order to identify attacks that steal data stored.

Formal method have already been applied for fault injection and simulation of the system after the occurrence of faults (among which [5,11]). In this paper, we propose a similar approach for the analysis of system security.

A CPS basically consists of a physical plant and a feedback controller. Attacks to CPSs include physical attacks on sensor devices or actuators, attacks on communications; and attacks on computing components. In this work, only attacks on sensors and actuators by manipulation of the sensor measurements provided to the controller, and by the alteration of data sent from the controller to the actuators are considered. These attacks affect the physical processes of the CPS. Two important parameters of the attack are (i) the time at which the attack occurs, because some states can

Dept. of Information Engineering, University of Pisa
Lgo. L. Lazzarino 1, I-56122 Pisa, Italy
Tel.: +39-050-2217541
E-mail: first.last@unipi.it

be more critical than others and an alteration of such states can lead to catastrophic failure; and (ii) the duration of the attack, since many attacks may have no adverse effects if they last only for a short time.

The Prototype Verification System (PVS) tool [31], a specification, verification, and simulation environment based on higher-order logic, is used for the specification of the control parts of CPSs. The physical parts are assumed to be described by other modelling tools. The effects of attacks to sensors/actuators are modelled by functions in the theory of the controller that change the values of data exchanged between the controller and sensors/actuators, according to the type of attack.

Then the INTO-CPS co-simulation framework [21] is used to generate simulation traces of the overall system. Results on a case study (a simple robot vehicle) are presented.

Co-simulation can be used to study the effect of the attacks on the dynamic of the complete CPS. Interesting properties of the CPS can also be proved by the PVS theorem prover, by building a theory that describes the physical part at an abstraction level that retains the details useful for the proof, and using such theory with the theory of the controller. In particular, invariants can be proved using induction on the length of the co-simulation run.

The methodology is general and can be applied to other type of attacks. Moreover, our analysis can be used to identify weaknesses of the CPS system. It can be useful also for the definition of mechanisms for attack detection, attack tolerance, and the estimation of attack consequences.

The paper is organized as follows: Section 2 provides a review of related works; Section 3 briefly describes the PVS framework, and the co-simulation framework; Section 4 describes our methodology to formally model an attack; Section 5 shows how to perform formal analysis; Section 6 shows an application of the method, using a Line Follower robot as a case study (the theory of the line follower robot, the theories of the modeled attacks and results of the co-simulation); Section 7 presents results on formal verification; Section 8 concludes the paper.

2 Related work

A recent survey by Humayed et al. [16] reports on a large number of publications from the literature on CPS security and proposes a classification framework based on three orthogonal criteria: security, with the categories of threats, vulnerabilities, attacks, and controls; components, with the categories of cyber, physical, and cyber-physical components; and systems, with categories related to general system characteristics, such as architecture or application field.

A similar work by Alguliyev et al. [1] analyses and classifies existing research papers on the security of cyber-physical systems. The paper discusses the main difficulties and solutions for the estimation of the consequences of cyber-attacks by considering attacks modelling and detection, and the development of security architectures. A tree of types of attacks is proposed, that includes attacks on sensor/actuators devices, attacks on computing components, attacks on communications, and attacks on feedback.

Yampolskiy et al. [35] classify cyber-physical attacks according to two dimensions: the Target Domain (the element which is directly targeted by the attack), and the Effect Domain (the *victim* elements that suffer the effects of the attack). In a CPS, these two elements can differ, e.g., an attack to a sensor (target element), can change the position (victim element) of a vehicle. They propose an approach based on Data Flow Diagrams (DFD) and the STRIDE threat model.

Burmester et al. [10] describe a formal model for CPS security based on hybrid timed automata and the Byzantine fault model, using an international natural gas distribution grid as a case study.

Ferrante et al. [13] approach the issue of security requirements specification for embedded systems by defining a UML profile and developing an automatic process to generate system requirements from user requirements.

Mitchell et al. [26] present a model-based approach for modelling and analysis of attacks and countermeasures in CPSs based on a Stochastic Petri Nets (SPN) formalism. The approach is applied to an electrical grid.

Khalil [18] presents an approach based on attack trees and simulation of probabilistically timed physical attacks for vulnerability assessment of critical infrastructures.

Our approach differs from the previous ones since both simulation and formal proofs of CPS resilience to an attack can be performed in the adopted framework.

Much research has been carried out on model-checking and theorem-proving formal verification methods for CPSs. Among others, SpaceEx [15] and HySAT [14] are symbolic model checkers for hybrid systems; KeYmaera [34] is a theorem prover for hybrid systems based on differential dynamic logic that has been applied in different application fields from transportation [17] to robotic systems [27].

Among the languages and verification systems, higher-order logic can be successfully applied to CPSs for its expressiveness and versatility. Our work relies on PVS [31],

which defines theories in a higher-order logic language, and provides an integrated view of formal verification by theorem proving and simulation.

Recently, a theorethical framework for the estimation of the impact of attacks to sensor devices in CPSs has been presented by Lanotte et al. [19] using non-deterministic probabilistic labelled transition systems. The method is based on weak bi-simulation, which has been extended with time and the notion of equivalence up to a given tolerance, measuring the probability that the system and the system under attack have the same behaviour.

Differently from [19], in our work co-simulation results are used to compare experimentally the normal behaviour of the system with the behaviour under attack, and formal proofs are used to check if the two behaviours are equivalent (i.e., the attack has no effect), or to derive hypotheses under which equivalence is guaranteed. Moreover, attacks on actuators are also covered, in addition to attacks on sensors.

3 Background

3.1 The PVS Environment

The *Prototype Verification System* (PVS) [31] is an interactive theorem-proving environment whose users can define theories in a higher-order logic language and prove theorems with respect to them. The language of PVS is a purely declarative language, but its PVSio extension [29] can translate PVS function definitions into Lisp code, so that a PVS expression denoting a function application with fully instantiated arguments can be interpreted as an imperative function call. The PVSio extension includes input/output functions allowing the system prototype to interact with the user and the computing environment. Moreover, MisraC code can be automatically generated from PVS theories for automata [22,23], using the PVSio-web tool-set [30].

The PVS specification language provides basic types, such as Booleans, naturals, integers, reals, and others, and type constructors to define more complex types. The mathematical properties of each type are defined axiomatically in a set of fundamental theories, called the *prelude*. Among the complex types, the ones used in this work are *record* types and *predicate subtypes*.

A *record* is a tuple whose elements are referred to by their respective *field* name. For example, given the declarations:

```
wheels: TYPE = [#
  left: Speed,
  right: Speed #]
axle: wheels =
      (# left := 1.0, right := 2.0 #)
```

`axle` is an instance of type `wheels` and the expressions `left(axle)` and `right(axle)` denote the speeds of the left and right wheels of `axle`, respectively. Equivalent notations are `axle‘left` and `axle‘right`.

The *overriding* operator `:=` in a `WITH` expression redefines record fields. With the declarations above, the expression

```
axle WITH [ left := -1.0 ]
```

denotes the record value `(#-1.0, 2.0#)`.

An example of predicate subtype is the following:

```
LightSensorReading: TYPE =
        { x: nonneg_real | x <= 255 }
```

which represents the real numbers in the $[0, 255]$ interval.

The PVS syntax includes the well-known logical connectives and quantifiers, besides some constructs similar to the conditional statements of imperative languages. These constructs are the `IF ... ENDIF` expression and the `COND ... ENDCOND` expression. The latter is a many-way switch composed of clauses of the form *condition* $\rightarrow$ *expression* where all conditions must be mutually exclusive and cover all possible combinations of their truth values (an `ELSE` clause provides a catch-all). The PVS type checker ensures that these constraints are satisfied.

Definitions within a given theory may refer to definitions from other theories. This makes it possible to build complex system specifications in a modular and incremental way. Theory `control_th` below imports `robot_th` and defines functions for controlling the robot.

```
robot_th: THEORY
BEGIN
 id: posnat
 State: TYPE [# ... #]
 ....
END robot_th

control_th: THEORY
BEGIN IMPORTING robot_th
 ACC_STEP: Speed = 0.1
 accelerate(st: State): State
 BRAKE_STEP: Speed = 0.05
 brake(st: State): State
 ...
END control_th
```

The PVS environment includes the NASALIB theory libraries [12] providing axioms and theorems addressing many topics in mathematics, including real number analysis, and it can be applied to model both the discrete and the continuous part of the system [6].

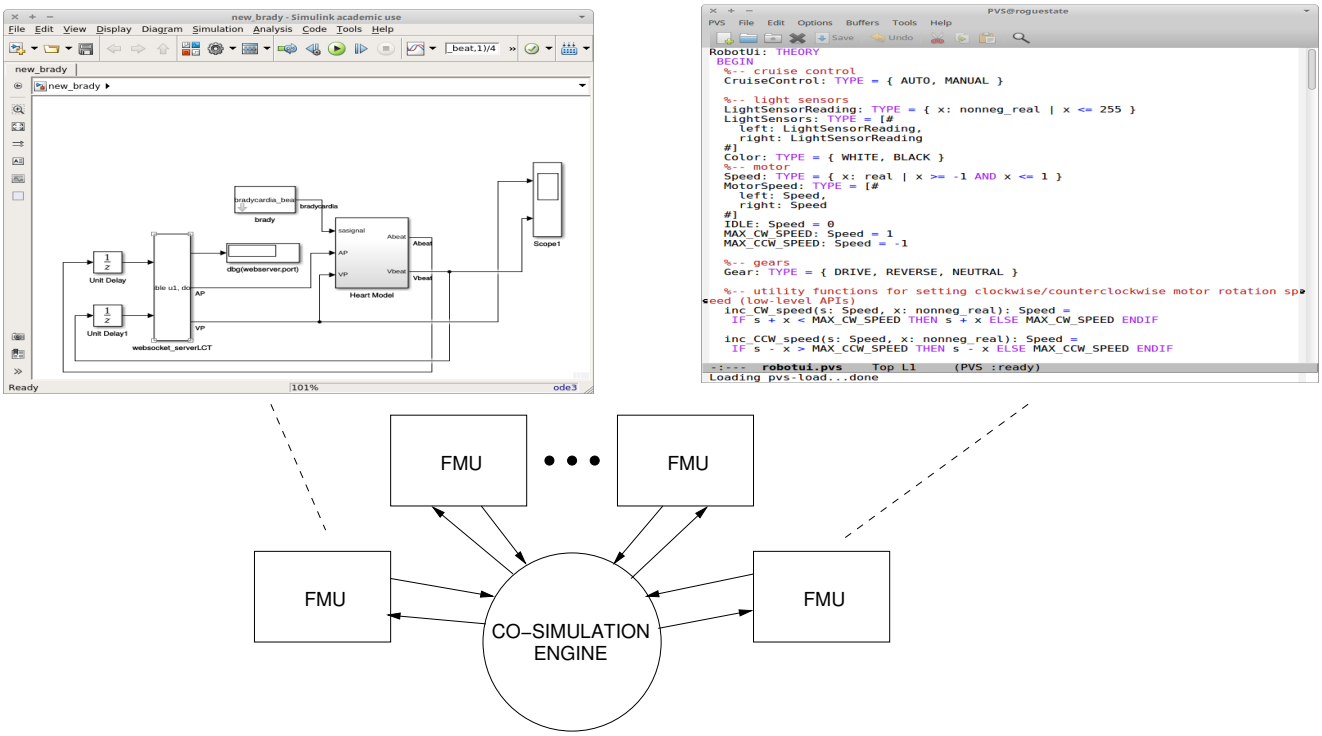

Fig. 1 FMI architecture.

The PVS theorem prover is based on the sequent calculus [32]. The structure of a *sequent* is in the following form, where the turnstile symbol '`|---`' separates the *antecedent* formulae above it from the *consequents* below.

```
{-1} A1
 ...
{-n} An
|-------
{1} B1
 ...
{m} Bm
```

A sequent is proved if (i) any consequent B_i is true, or (ii) any antecedent A_i is false, or (iii) any formula occurs both as an antecedent and as a consequent. The proof of a sequent consists in applying various inference rules until one of the above sequent forms is obtained. A formula to be proved is represented as a sequent without antecedents.

3.2 The Co-simulation Framework

Co-simulation is the joint simulation of independent sub-models each representing a component or subsystem of the overall system.

In the FMI standard [8], co-simulation is performed by a number of *Functional Mockup Units* (FMUs), each responsible for simulating a single model in the native formalism with the tool used to create the model. An FMU is a wrapper that executes commands from an orchestration process, called a Co-Simulation Engine (COE), interacting with the simulation software encapsulated in the FMU, in this case the PVSio interpreter. The COE communicates with the FMUs to exchange data, in a master-slave configuration. The FMI architecture is shown in Figure 1.

The COE and the FMU exchange commands and data using buffers in the FMU. The COE invokes (i) `fmi2Set()` to update the values of the input variables in the buffers of the FMU; (ii) `fmi2DoStep()` to execute a co-simulation step; and (iii) `fmi2Get()` to get the new values of the output variables from the buffers of the FMU.

A PVSio-based FMU implements `fmi2DoStep()` to copy the values of the input variables from the FMU buffers to the PVSio state; invokes the simulator and copies the values of the output variables from the PVSio state to the buffers.

INTO-CPS [4,21] is a co-simulation environment that integrates tools for the engineering of cyber-physical systems, covering both modeling of discrete and continuous behaviors and formal proofs.

Examples of tools available in INTO-CPS for modeling and analysis are Modelio [28], Overture [20], and 20-sim [9].

In [33], the authors extended the INTO-CPS co-simulation framework with FMUs based on the PVSio

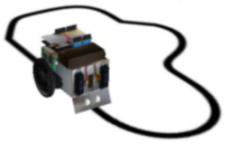

Fig. 2 The INTO-CPS Line Follower robot case study.

tool. Such FMUs can also implement a user interface using PVSio-web [30], allowing user interaction in the co-simulation.

With the approach used in this work, the PVS language is used to model the control part of a CPS, while the plant components are modeled with other industry-standard languages such as Simulink, 20-Sim, or Modelica. Each of these environments can produce an FMU. In particular, the FMU for the PVS theory is produced with the PVSio-web development tool, introduced in the previous subsection.

Fig. 3 Architecture of interactive FMUs.

Let us consider the INTO-CPS Line Follower robot case study (http://projects.au.dk/into-cps/). The Line Follower robot is a small vehicle that can follow a path defined by a black line painted on a white floor. Figure 2 shows a practical realization of the robot.

Figure 3 shows the co-simulation framework for the INTO-CPS Line Follower robot case study with automatic and manual control.

The FMU for the controller of the robot is modeled in PVS and the user interface for the robot is a joystick rendition, i.e., a picture of a real device with PVS functions assigned to widgets and displays. A PVSio-web module interprets user interactions as messages sent to the FMU, which executes the corresponding functions.

The user can acquire control of the robot from the joystick, manually control the robot with buttons, and switch the robot back to automatic control. On the right-hand side of the joystick, the path followed by the robot is shown. On the left-hand side of the joystick, information on the velocity and direction of the robot are reported (the upper "D" stands for the gear (i.e., direct, reverse or neutral).

In the developed environment, a real joystick could also be used in the co-simulation instead of the virtual one. The Line Follower robot case study is better explained in Section 6.

4 Modeling systems and attacks

The behavior of a cyber-physical system relies on a control loop, designed to implement the desired control laws. At each cycle of the loop, sensors in the plant send data to the controller, which acts on the plant sending commands to the actuators.

In the FMI framework, the controller and the plant are FMUs, and the COE links outputs of the Plant FMU with inputs of the Controller, and viceversa.

4.1 Controller theories

A key feature of the PVS environment is the possibility of specifying a system in a way that is both purely formal and executable. This is made possible by the PVSio ground evaluator, which, as hinted to in Section 3.1, can interpret a theory and interact with the computing environment.

A CPS controller can be specified with a theory with the following schema:

- Definitions of application-specific variable or constant magnitudes.
- A data structure representing the instantaneous state of the controller, typically including values read from sensors, values sent to actuators, time information, and other possible data.
- One or more functions to update the state at each discrete time step.
- An initial state.
- A time-advancement function that recursively updates the state.

The specification consists of two basic elements: the state of the sub-system (`State`) and the function `tick(State)`, which given a state, according to control

laws, computes the output to be forwarded to other sub-systems.

4.2 Attack theories

In this work, we consider the following types of attacks:

- *Attack to sensors.* The effect of such an attack is the corruption of data read from sensors. At the beginning of each co-simulation step, such data are stored into the input variables of the controller's state.
- *Attack to actuators.* The effect of such an attack is the corruption of data sent to actuators. At the end of each co-simulation step, such data are stored into the output variables of the controller's state.

The FMU of the control part is modified as follows:

- Each attack is modeled by a function that alters the system state according to the attack's envisioned effects.
- For each attack, the time of occurrence of the attack must be specified, distinguishing between permanent attacks and temporary attacks, and, in the latter case, distinguishing between sporadic attacks and attacks executed only once.
- An attack can be simulated in two ways: (i) It can be generated internally by the simulation algorithm, or (ii) it can be activated interactively by the user.

In particular, an attack A is formally specified by a set of state variables, a set of clocks and a set of guarded statements:

$A = \langle Var_A, Clk_A \cup \{stepCounter\}, Com_A \rangle$

- *State variables.* Var_A is the set of variables of the state of the controller that are accessed by the attacker.
- *Clocks.* Two types of clocks are used: a set Clk_A of attacker clocks and a global clock *stepCounter*, which is initialized to 0 when a co-simulation run starts and is incremented for each co-simulation step. The attacker cannot modify this global clock.
- *Guarded statements.* Com_A is a set of guarded statements. A guarded statement has the form: $[condition \rightarrow x_1 := v_1; \cdots; x_n := v_n]$, where *condition*, the guard of the statement, is a condition on clocks (using logical operators $\wedge$, $\vee$, $=$, $\neq$) and the statement is a sequence of assignments to state variables or to local clocks ($x_i := v_i$; with $x \in Var_A \cup Clk_A$). Guards must be mutually exclusive.

To model attacks, we extend the state of the system in the controller with the `stepCounter` and with the set of local clocks `Clock_A`.

The effects of the attack are described by a function in PVS, whose skeleton is described below:

```
fun_attack(st: State): State =
    IF condition
       THEN st
         WITH [x1 := v1,
               ...,
               xn := vn
              ]
       ELSE st
    ENDIF
```

Some possible attacks could be:

- every 20 simulation steps, increment by 3 the value read from a sensor;
- every 100 simulation steps, lock at zero the value sent to an actuator for 20 steps;
- double the value of a sensor randomly in the co-simulation.

A local clock is used to count the number of steps between two attacks; and two local clocks are used to model the lock-at-zero attack, one to count the duration of the attack and another to count the steps between two attacks.

Probabilistic behaviors can be encoded in attacks using the function `NRANDOM(n: posnat)`, which is available in the PVS framework and that implements a uniform pseudo-random number generator that returns a natural number in the interval `[0::n)`. Using the language of PVS, more sophisticated attacks could also be implemented.

The control algorithm comprises a set of pre-defined functions to model different types of attacks. These functions can be executed in two modes, programmed or interactive.

4.3 Programmed Attacks

An attack has an initial time and a duration both of which may be hardcoded or random.

Let `Sensor_attack` be a function modeling an attack to sensors. The behavior of the system under attack is specified as the result of the function `tick()` on the extended state of the system after the attack to sensors:

```
system_under_attack(st: State) :
                              State =
      LET st1 = Sensor_attack(st),
          IN tick(st1)
```

We assume that `State` is the state of the system with the addition of a variable for each clock defined in the model of the attacks, and `tick()` is the function applied by the controller. The `LET ... IN ...` construct

introduces a definition to be used in the expression following `IN`.

If `Actuator_attack` is a function modeling an attack to actuators, the behavior of the system under attack is specified as the result of the `Actuator_attack` function on the state of the system generated by function `tick()`.

```
system_under_attack(st: State) :
                            State =
      LET st1 =  tick(st)
          IN Actuator_attack(st1)
```

Finally, the two attacks could be combined. Since attacks to sensors affect the inputs to the controller and attacks to actuators affect its outputs, `system_under_attack` first passes the current state to the function modeling sensor attacks, then the resulting state is passed to the controller, which computes another state that is further transformed by the function modeling actuator attacks, as shown in the following code.

```
system_under_attack(st: State) :
                            State =
      LET st1 = Sensor_attack(st),
          st2 =  tick(st1)
          IN Actuator_attack(st2)
```

4.4 Interactive Attacks

Interactive attacks are activated and controlled by a simulation user playing the role of the adversary. This is implemented through the definition of a predefined set of attacks as functions in the PVS theories, and the creation of button widgets on the graphical interface, one widget for each attack function (Figure 4). An attack starts when a user clicks the button. The function linked to the attack is invoked at each co-simulation step. The attack terminates when the user clicks "Stop". Attacks can be executed in sequence during a co-simulation run, but only one attack can be active at a time.

This implementation uses the PVSio-web [30] tool, which allows us to create the graphical interface of a device and to link interface elements with functions describing how the device responds to user actions.

When an action is executed (e.g. user clicks a button), a JavaScript module sends the appropriate command to the PVS FMU that executes the action in the co-simulation step. In our case, the command is "execute `fun_attack_id()` before `tick()`" or "execute `fun_attack()_id` after `tick()`" depending on the type of attack.

Interactive attacks have some limitations:

– when an attack is activated interactively, it is independent of the current co-simulation timestep, and

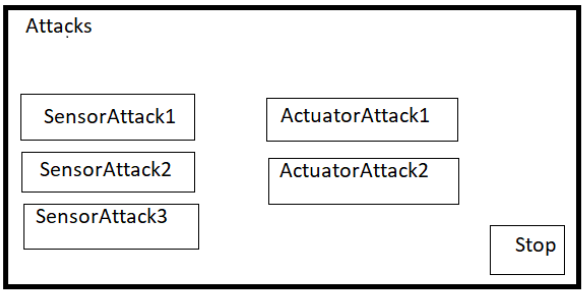

Fig. 4 A skeleton of user interface for interactive attacks.

– each sensor/actuator can be assigned distinct values, but this values are fixed and cannot be changed during the attack.

When the user clicks "Stop", the simulated system is no longer under attack, until a new attack is activated. As a consequence, it is possible to change the number and the duration of the attacks during a co-simulation run. Further improvements to overcome the limitations are under development.

5 Analysis of system behaviour

The framework proposed in [33] to generate an FMU starting from a PVS theory enables the usage of the co-simulation for validation of the system under attack. Modelling attacks in PVS enables the usage of the theorem prover for verification. Example of verification and validation will be applied on the case study.

5.1 Co-simulation for validation

Function `fmi2DoStep()` invokes the transition function `system_under_attack(State)`. Assuming the system in under sensor attack, values set in the input variable buffer by the `fmi2Set()` function will be corrupted by the attack function before the invocation of the `tick()` function. Likewise, if the system is under actuator attack, the changes of the state produced by the `tick()` function will be corrupted by the attack function before the invocation of the `fmi2Get()`.

In both cases, data sent from the controller FMU to the plant FMU will suffer the effects of the attack thus affecting the whole co-simulated system. Users can run the co-simulation to validate the effects of the attack; if users create an attack that is expected to hinder the performance of the system then, they need to compare the co-simulation of the original system against the system under attack and check that the performance of the

latter is worse. Through the co-simulation it is possible to check also if the system stops at the specific time.

Users have three different methods to analyse the results of the co-simulation: (i) inspect the PVSio-web user interface; (ii) inspect the graph generated at runtime by the INTO-CPS application (for example, Figure 7 shows the speed of the robot), or (ii) inspect the log file produced by INTO-CPS at the end of the co-simulation for more detailed information.

5.2 Formal verification of system under attack

Formal proofs can be used to verify key properties of the control system under attack. In particular, an invariant is a property that must be proved for all states of all possible execution traces. This part only applies to programmed attacks because interactive attacks rely on the non deterministic behaviour of the user, which demands a different approach. An execution trace of the system is a sequence of states that starts with an initial state and applies the state-transition function to generate subsequent states. The following function `kth_step` can be used to define all sequences of states given by all possible executions.

```
kth_step(K: nat): RECURSIVE State =
  IF (K = 0) THEN init_state
  ELSE system_under_attack(kth_step(K-1))
  ENDIF
  MEASURE K
```

Function `kth_step` takes parameter `K` and recursively applies `K` steps of the state-transition function `system_under_attack`. In PVS, the termination of the recursion has to be demonstrated, and the `MEASURE` part provides such information to the type checker and prover.

Function `kth_step` is convenient when proving invariants since it allows building the proof by induction on the length of the trace. For instance, the proof of an invariant `P` can be expressed by the following theory:

```
TH1: THEOREM
  FORALL(K: nat):
      P((kth_step(K))
```

PVS provides support for different induction schemes, e.g., classical induction, or structural induction on graphs and paths.

Users can enable the verification of the whole system (the plant and the controller) by expressing the plant model in PVS. Usually an abstract model of the plant is built. The abstract model must contain enough details to keep the difference between the original system and the abstract one below an acceptance threshold of tolerance for the properties that must be proved. An example of abstract model is shown in Section 7.2, where the time-continuous kinematics of the Line Follower robot are modelled with a discrete-time PVS theory, assuming a constant angular and linear speeds of the robot for every Δ discretization time.

6 A case study

The system considered in this work is the Line Follower robot case study of the INTO-CPS project[1], see Figure 2 in Section 3.

The robot has two drive wheels each propelled by its own independent motor, and two optical sensors, symmetrical with respect to the longitudinal axis, that measure the reflected light intensity of the floor immediately ahead of the robot. The robot starts astride the black line, so that both sensors see the white floor. The robot keeps heading forward as long as both sensors detect a white color. When the path curves, one sensor intercepts the black line while the other still sees the white floor.

The robot controller then steers the vehicle by slowing down the internal wheel (on the side of the sensor detecting the line) with respect to the external one. In addition to the automatic mode of operation, it is possible for an operator to override the automatic control and drive the robot with a remote dashboard (see Figure 3 in Section 3).

This work considers the automatic mode of operation, identified by the string `AUTO` on the graphical interface (top of Figure 3).

6.1 Robot Theory

In the following, we show the main parts of a PVS theory describing the above system.

First, some type definitions provide the types of data needed for the model: `CruiseControl` to distinguish the two modes of operation, `LightSensorReading` to specify the values from the sensors, `LightSensors` to access the left and right sensor readings, `Speed` to specify the angular speed range for the wheels, and `MotorSpeed` to control the two wheel motors, and `Gear` to distinguish the three modes of the gear train. Positive and negative speed values represent clockwise and counterclockwise rotation, respectively.

```
robotUI: THEORY
BEGIN
```

[1] http://projects.au.dk/into-cps/

```
  CruiseControl: TYPE = { AUTO, MANUAL }

  LightSensorReading: TYPE =
     { x: nonneg_real | x <= 255 }

  LightSensors: TYPE = [#
    left: LightSensorReading,
    right: LightSensorReading
  #]


Speed: TYPE =
       { x: real | x >= -1 AND x <= 1 }
  MotorSpeed: TYPE = [#
    left: Speed,
    right: Speed
  #]
```

Data of the above types compose the system state.

```
  State: TYPE =
        [# lightSensors: LightSensors,
           motorSpeed: MotorSpeed,
           gear: Gear,
           time: real,
           cc: CruiseControl,
           % ....fields for attacks
        #]
```

The control algorithm is specified by functions that update the system state by setting the motor speed depending on the sensor readings. In the two following functions, a value of `150` units is chosen as the threshold *thr* between a high (white) and a low (black) light intensity. Note that for each combination of readings, the two motors have opposite directions, due to the mechanical arrangement.

```
 thr: LightSensorReading = 150
 low: Speed = 0.1
 med: Speed = 0.4
 hi: Speed = 0.5
 update_left_motor_speed(st: State): Speed =
  LET ls = lightSensors(st) IN
  COND ls`right <= thr AND ls`left <= thr -> med,
       ls`right <= thr AND ls`left > thr -> hi,
       ls`right > thr AND ls`left <= thr -> low,
       ELSE -> motorSpeed(st)`left
  ENDCOND

 update_right_motor_speed(st: State): Speed =
  LET ls = lightSensors(st) IN
  COND ls`right <= thr AND ls`left <= thr -> -med,
       ls`right <= thr AND ls`left > thr -> -low,
       ls`right > thr AND ls`left <= thr -> -hi,
       ELSE -> motorSpeed(st)`right
  ENDCOND
```

The simulation is driven by a `tick()` function that is called at each simulation step to update the motor speeds and increment time:

```
 tick(st: State): State =
   IF cc(st) = AUTO
   THEN st WITH [motorSpeed := (#
     left := update_left_motor_speed(st),
     right := update_right_motor_speed(st)#),
     time := time(st) + 0.01 ]
   ELSE st WITH [time := time(st) + 0.01]
   ENDIF
```

where `init_state` is a constant of type `State` defining the initial state.

Finally, the theory defines functions (not shown) called from the user interface to switch between automatic and manual control, and in the latter case to execute user requests, such as accelerating, decelerating, or steering.

Figure 5 shows the results of the co-simulation (blue line) superimposed on the expected path (gray line) when the system is co-simulated for 20 s, assuming a co-simulation step of 0.01.

6.2 Attack Theories

An attack is injected into the system by executing the controller together with the functions modeling attacks. In order to model attacks, the robot state is extended with fields characterizing the different types of attacks.

In the present example, three attacks are considered: an attack to sensors that occurs once and acts indefinitely; another attack to sensors that repeated N number of steps ever M steps; and attack to actuators that occurs sporadically with a duration of one co-simulation step.

The following function implements an attack that indefinitely forces to *black* the value read by the left sensor, starting from a randomly chosen co-simulation step. Function NRANDOM in the initial state is invoked with an upper bound of 500. Variable `lightSensors` is modified (`140` is the constant for black color); clock $step\_of\_attack$ specifies the co-simulation step at which the attack starts. The attack is defined as:
$Var = \{lightSensors\}$
$Clock = \{start\_step, stepCounter\}$
Com is the body of the following function.

```
attack1(st: State): State = % sensor attack
    IF stepCounter(st) > start_step(st)
    THEN st WITH [
        lightSensors :=
            (# left := 140,
               right := st`lightSensors`right #)
        ]
    ELSE st
    ENDIF
```

The following function implements an attack that forces to *white* the value read by the left sensor for

`L` steps every `M` timesteps. This is repeated indefinitely, starting from the first co-simulation step. Variable `lightSensors` is modified (`160` is the constant for white color); clock *elapsed_steps* specifies the number of steps since the last attack has been activated. The attack is defined as:
$Var = \{lightSensors\}$
$Clock = \{elapsed\_steps, stepCounter\}$
Com is the body of the following function.

```
attack2(st: State): State = % sensor attack
    IF elapsed_steps(st) <= L
    THEN st WITH [
        lightSensors :=
            (# left := 160,
               right := st`lightSensors`right #),
        elapsed_steps :=  elapsed_steps + 1
        ]
    ELSE IF elapsed_steps(st) < M
          THEN st WITH [
             elapsed_steps :=  elapsed_steps + 1
           ]
           ELSE IF elapsed_steps(st) = M
                THEN st WITH [
                     elapsed_steps := 0 ]
    ENDIF
```

The following function implements an attack that sporadically switches off the power of each motor for one co-simulation step. The co-simulation step, at which the power of each motor is switched off, is chosen randomly. Function NRANDOM is invoked in the initial state and in the attack function with an upper bound of 20. Clock *occurrence_step* specifies the co-simulation step at which the next occurrence of the attack starts. The attack is defined as:
$Var = \{motorSpeed\}$
$Clock = \{occurrence\_step, stepCounter\}$
Com is the body of the following function.

```
attack3(st: State): State =  % actuator attack
   IF stepCounter(st) = occurrence_step(st)
   THEN st WITH [
        motorSpeed :=
            (# left := 0,
               right := 0 #),
               occurrence_step := NRANDOM(20) + 1
                ]
   ELSE st
   ENDIF
```

The full definition of *State*, including information about the attacks above, is the following:

```
  State: TYPE =
       [# % ... robot state

          % global clock
           stepCounter:int,

          %local clocks
          start_step: int, % attack1
          elapsed_steps: int,  % attack2
          occurrence_step: int  % attack3
       #]
```

Fig. 5 No attack.

In the initial state, the step at which attack1 start and the the step at which the first occurrence of attack3 starts are initialized with a random value; the elapsed steps since the last occurrence of the attack2 is initialized to 0:

```
init_state: State =
    (# % ... robot state

      % global clock
      stepCounter := 0;

      % attack1
      start_step = NRANDOM(500),

      % attack2
      elapsed_steps = 0,

       % attack3
       occurrence_step = NRANDOM(20) + 1
    #)
```

Function `tick()` implements the controller as previously shown in Section 6.1, except that it also updates the global clock (`stepCounter`).

```
tick(st: State): State =
  IF cc(st) = AUTO THEN
     % ... omitted
     stepCounter := stepCounter + 1;
    ELSE
     % ... omitted
     stepCounter := stepCounter + 1;
 ENDIF
```

6.3 Execution Traces

Figure 6 shows the sample trajectory when the attack3 to actuators occurs. The robot follows the nominal path, but the execution traces, reporting the simulated time at each simulation step, show that the robot is delayed with respect to the resulting trace shown in Figure 5. This is expected, since the attack consists in stopping

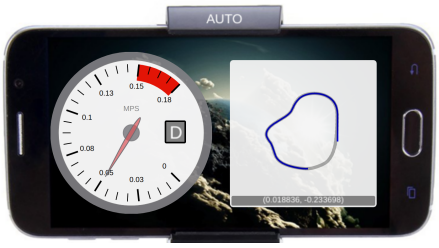

Fig. 6 Attack to actuators (attack3).

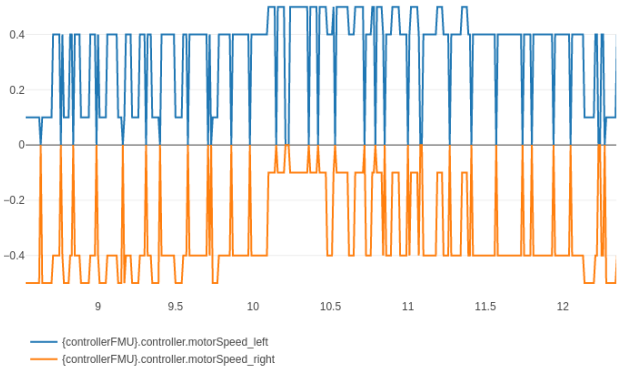

Fig. 7 INTO-CPS graph of left motor (blue line) and right motor (red line) speed under attack3.

Fig. 8 Attack to sensors: attack1.

both motors for a short time. Since the motors stop at the same time, the robot heading at each instant is unchanged.

Figure 7 shows the two outputs of the control FMU during a co-simulation run: according to the actuator attack function the values on the graph sporadically go to zero.

Figure 8 show the sample trajectory for the attack1 to sensors. The left sensor is stuck at a fixed value, so that the robot starts turning at the onset of the attack, ending up in a closed trajectory.

Figure 9 shows the effect of the attack2 to sensors, assuming `L = 40` and `M = 2*L`. At a given point, the robot looses the line. The same attack under different values of `L` has no effect. In case of `L = 20`, the trajectory of the robot is equivalent to the one shown in Figure 5. This example is better investigated in Section 7.3.

Fig. 9 Attack to sensors: attack2.

Fig. 10 User interface for interactive attacks of LFR.

6.4 Interactive attack theory

The attacks formalized in the previous section can also be performed interactively by combining the functions shown below with the user interface shown in Figure 10. *ActsZero* (Actuators to zero) forces the motor speed to zero and *SLBlack* (Sensor Left Black) forces the left sensor to a value above the threshold.

```
SLBlack(st:State): State = st WITH [
  lightSensors := (#
    left:=140,
    right:=st‘lightSensors‘right
    #)
]

ActsZero(st:State): State = st WITH [
  motorSpeed := (# left:=0,right:=0#)
  ]
```

Figure 10 shows a co-simulation run where the user (i) has pushed the SLBlack button after the start, forcing the robot to turn left, and then (ii) has pushed the Stop button to stop the attack and successively has pushed the ActsZero button while the robot is the middle of the shape, far away from the line.

7 Formal verification

In this section some invariants of the system are proved.

7.1 A property of the Controller

As an example of a possible verification on the controller's behavior, this section shows results concerning attacks causing a sensor to be stuck at a given value.

In particular, the following theorem shows that, under an attack forcing the left sensor to read a "black" value (i.e., greater than *thr*), the robot can never turn right (i.e. motorSpeed of the right wheel will always be greater or equal than motorSpeed of the left wheel).

```
N: above(1)
never_right_random: THEOREM
    FORALL(K: above(NRANDOM(500) + N)):
        motorSpeed(kth_step(K))‘left
        <= -motorSpeed(kth_step(K))‘right
```

The proof is inductive on the number of steps K. The prover's *induct* rule generates the induction base and the inductive step:

```
Rule? (induct K)
Inducting on K on formula 1,
this yields  2 subgoals:
never_right_random.1 :

  |-------
{1} motorSpeed(
      kth_step(NRANDOM(500) + N + 1))‘left <=
     -motorSpeed(
      kth_step(NRANDOM(500) + N + 1))‘right
  .
  .
  .
never_right_random.2 :

  |-------
{1} FORALL (ja: above(NRANDOM(500) + N)):
        motorSpeed(kth_step(ja))‘left <=
        -motorSpeed(kth_step(ja))‘right
        IMPLIES
        motorSpeed(kth_step(ja + 1))‘left <=
        -motorSpeed(kth_step(ja + 1))‘right
```

Both subgoals are proved with a lengthy but obvious sequence of function expansions, introduction of a small number of intermdiate lemmas (not shown), and automatic simplifications with the *simplify* rule, closed by invocations of the *assert* and *grind* rules that conclude the subproofs.

7.2 A theory for the robot kinematics

In the preceding sections, the robot kinematics have been simulated by an FMU encapsulating a 20-sim model, and formal verification has addressed only the controller model. In this section, a PVS model of the robot kinematics is introduced. This model extends the original definition of *State* by introducing new fields for co-simulation: step size (*stepsize*), linear and angular speed (*linspeed* and *angspeed)*, position (coordinates *xx* and *yy*, and direction (angle *theta*).

```
extended_robot: THEORY
 BEGIN
 IMPORTING RobotUi

 ext_State: TYPE =
        [#  state: State,
            stepsize: real,
            linspeed: real,
            angspeed: real,
            xx: real,
            yy: real,
            theta: real
 #]
```

The model also extends the previous definition of *tick* introducing two new functions: *update_position* and *update_speed*.

```
ext_tick(st: ext_State): ext_State =
  update_position(update_speed(st))

update_position(st: ext_State): ext_State =
  st WITH
[
xx:= st‘xx-st‘linspeed*st‘stepsize*SIN(st‘theta),
yy:= st‘yy+st‘linspeed*st‘stepsize*COS(st‘theta),
theta := st‘theta+st‘angspeed*st‘stepsize
  ]

update_speed(st: ext_State): ext_State =
  LET st1: ext_State =
    st WITH [state := tick(st‘state)] IN
     st1 WITH [
       linspeed := COND
    st1‘state‘motorSpeed‘left = 0 -> 0,
    st1‘state‘motorSpeed‘left = med -> 0.062,
    st1‘state‘motorSpeed‘left = hi -> 0.057,
    st1‘state‘motorSpeed‘left = low -> 0.057,
    else -> st1‘linspeed
       ENDCOND,
       angspeed := COND
    st1‘state‘motorSpeed‘left = 0 -> 0,
    st1‘state‘motorSpeed‘left = med -> 0,
    st1‘state‘motorSpeed‘left = hi -> -0.47,
    st1‘state‘motorSpeed‘left = low -> 0.47,
    else -> st1‘angspeed
       ENDCOND
       ]
```

```
END extended_robot
```

Function *update_speed* invokes the *tick* function and then computes the linear and angular speeds. Function *update_position* updates the position and the direction of the robot based on the linear and angular speeds.

The formulas used in these functions are a simplified version of the formulas used in the 20-sim models. The simplification consists of introducing instant changes of both the linear speed of the robot and the angular one. The constant values in function *update_speed* are taken from a co-simulation log and they represent: the maximum linear speed when the robot is moving forward (0.062 m/s), the maximum linear speed when the robot is turning (0.057 m/S) and the angular speed when the robot is turning (0.47 rad/s).

Using the maximum speed, we obtain an over-approximation of the distance covered by the robot. This abstract model can be used to prove properties such as the one described later in this section, where the maximum distance covered in a co-simulation step is used in the proof; if a property is satisfied using the abstract model, the property holds also on the real system.

The new theory has been embedded into an FMU and co-simulated for validation: the result is shown in Figure 11 where the temporal evolution of the y and x coordinates generated with the 20-sim model (blue lines) are compared with the ones of the PVS model (red lines); the behaviour is the same within a tolerance of approximately 5 centimeters.

PVS allows us to describe a system at different levels of abstractions. Adding more details to the PVS kinematics model finer properties could be proved and better tolerance could be achieved.

7.3 A property of the complete CPS

The usage of the kinematic theory enables the proof of properties related to the physical process. Let us consider the attack2 described in Section6.2, with

- `L` a value that represents a generic number of co-simulation steps
- `S` a value for the step size (*stepsize*)
- 1.5 centimeters (0.015 meters) the width of the black line painted on the floor of the robotic system.

Under the hypothesis that `L * S <= 0.24` it can be proved that the difference between the initial and final y coordinate of the robot is less than or equal to 0.015 meters (the width of the line).

The property can be expressed in PVS as follows:

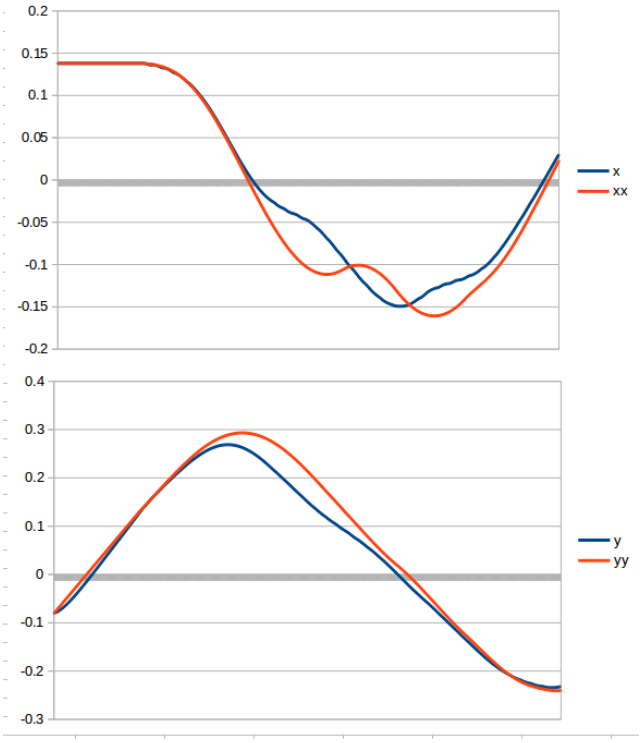

Fig. 11 Comparison between 20-sim model and PVS model.

```
maximum_step_attack: THEOREM
 L*S <= 0.24 IMPLIES
  FORALL(K:above(L)):
    kth_step(K)‘yy - kth_step(K-L)‘yy <=0.015
```

The theorem states that if `L*S <=0.24` then within `L` steps the y (x) coordinate of the robot change at most of 1.5 centimeters (0.015 meters) which is the width of the line in the current case study.

The proof is inductive on the number of steps K, using the same commands used for the theorem in the previous section

```
[-1] L * S <= 6/25
  |-------
{1} kth_step(K)‘yy -
        kth_step(K - L)‘yy <=  3/200
```

In the sequent, `[-1]` is the antecedent and `{1}` is the consequent. The property holds independently of the initial position of the robot on the line, only the constraint on the duration of the attack (`L * S`) must be satisfied. A similar theorem can be proved on the x coordinate.

This theorem can be exploited in the design of attacks: if an attack changes the value of the left sensor to white for less than `L * S` time, then the attack will

never move the robot from one side of the line to the other. This means that if a security system is able to detect an attack within `L` steps, it manages to keep the robot close to the line.

If the hypothesis of the theorem is not true, we do not have information. Figure 9 shows a co-simulation run with `L = 40` and `M = 80` and `S = 0.01`. Since $L * S = 0.4 > 0.24$, the hypothesis of the theorem is not met. In this case, the left sensor moves from the internal side of the path to the external side and the robot drives away from the painted line.

8 Conclusions

This paper reports on our work in defining a methodology to model attacks and analyzing the effects of security attacks in cyber-physical systems using a co-simulation framework and formal verification. We restrict our attention to attacks on sensors and actuators. As further work, other categories of attacks will be considered and we intend to improve the formalisation of attacks using timed automata [2] and the translation from networks of timed automata to PVS theories defined in [7].

Moreover, the framework allows formal proofs of invariants on the state of the system through theorem proving. Examples of properties of the system under attack that are satisfied for all co-simulation runs have been shown. The level of abstraction of the physical process requires a careful consideration, and we intend to investigate strategies to generate abstract models that retain sufficient information to enable the proof of some type of properties.

Acknowledgements

The authors also thank the INTO-CPS project for providing the case study and the co-simulation environment.

References

1. Alguliyev, R., Imamverdiyev, Y., Sukhostat, L.: Cyber-physical systems and their security issues. Computers in Industry **100**, 212 – 223 (2018). DOI https://doi.org/10.1016/j.compind.2018.04.017
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (1994)
3. Avvenuti, M., Bernardeschi, C., Francesco, N.D., Masci, P.: JCSI: A tool for checking secure information flow in java card applications. Journal of Systems and Software **85**(11), 2479–2493 (2012). DOI 10.1016/j.jss.2012.05.061
4. Bagnato, A., Brosse, E., Quadri, I., Sadovykh, A.: INTO-CPS: An integrated "tool chain" for comprehensive model-based design of cyber-physical systems (2015). This publication is part of the Horizon 2020 project: Integrated Tool chain for model-based design of CPSs (INTO-CPS), project/GA number 644047.
5. Bernardeschi, C., Cassano, L., Domenici, A., Sterpone, L.: ASSESS: A simulator of soft errors in the configuration memory of SRAM-Based FPGAs. IEEE Trans. on CAD of Integrated Circuits and Systems **33**(9), 1342–1355 (2014). DOI 10.1109/TCAD.2014.2329419
6. Bernardeschi, C., Domenici, A.: Verifying safety properties of a nonlinear control by interactive theorem proving with the Prototype Verification System. Inf. Process. Lett. **116**(6), 409–415 (2016). DOI 10.1016/j.ipl.2016.02.001
7. Bernardeschi, C., Domenici, A., Masci, P.: A PVS-Simulink Integrated Environment for Model-Based Analysis of Cyber-Physical Systems. IEEE Trans. Software Eng. **44**(6), 512–533 (2018). DOI 10.1109/TSE.2017.2694423
8. Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauß, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., Olsson, H., Viel, A.: Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In: Proceedings of the 9th International MODELICA Conference;September 3-5; 2012; Munich; Germany, no. 76 in Linköping Electronic Conference Proceedings, pp. 173–184. Linköping University Electronic Press (2012). DOI 10.3384/ecp12076173
9. Broenink, J.F.: 20-SIM software for hierarchical bond-graph/block-diagram models. Simulation Practice and Theory **7**(5), 481–492 (1999). DOI https://doi.org/10.1016/S0928-4869(99)00018-X
10. Burmester, M., Magkos, E., Chrissikopoulos, V.: Modeling security in cyberphysical systems. International Journal of Critical Infrastructure Protection **5**(3), 118 – 126 (2012). DOI https://doi.org/10.1016/j.ijcip.2012.08.002
11. Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E. (eds.): Methods, Models and Tools for Fault Tolerance. Springer-Verlag, Berlin, Heidelberg (2009)
12. Dutertre, B.: Elements of mathematical analysis in pvs. In: Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs '96, pp. 141–156. Springer-Verlag, Berlin, Heidelberg (1996)
13. Ferrante, A., Kaitovic, I., Milosevic, J.: Modelling requirements for security-enhanced design of embedded systems (2014). DOI 10.5220/0005050003150320
14. Fränzle, M., Herde, C.: Hysat: An efficient proof engine for bounded model checking of hybrid systems. Formal Methods in System Design **30**(3), 179–198 (2007). DOI 10.1007/s10703-006-0031-0
15. 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: G. Gopalakrishnan, S. Qadeer (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV), no. 6806 in LNCS, pp. 379–395. Springer (2011). DOI 10.1007/978-3-642-22110-1_30
16. Humayed, A., Lin, J., Li, F., Luo, B.: Cyber-Physical Systems Security—A Survey. IEEE Internet of Things Journal **4**(6), 1802–1831 (2017). DOI 10.1109/JIOT.2017.2703172
17. Jeannin, J., Ghorbal, K., Kouskoulas, Y., Gardner, R., Schmidt, A., Zawadzki, E., Platzer, A.: A formally verified hybrid system for the next-generation airborne collision avoidance system. In: C. Baier, C. Tinelli (eds.)

TACAS, *LNCS*, vol. 9035, pp. 21–36. Springer (2015). DOI 10.1007/978-3-662-46681-0_2
18. Khalil, Y.: A novel probabilistically timed dynamic model for physical security attack scenarios on critical infrastructures. Process Safety and Environmental Protection **102**, 473 – 484 (2016). DOI https://doi.org/10.1016/j.psep.2016.05.001
19. Lanotte, R., Merro, M., Tini, S.: Towards a formal notion of impact metric for cyber-physical attacks. In: Integrated Formal Methods - 14th International Conference, IFM 2018, Proceedings, pp. 296–315 (2018). DOI 10.1007/978-3-319-98938-9_17
20. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative Integrating Tools for VDM. SIGSOFT Softw. Eng. Notes **35**(1), 1–6 (2010). DOI 10.1145/1668862.1668864
21. Larsen, P.G., Fitzgerald, J., Woodcock, J., Fritzson, P., Brauer, J., Kleijn, C., Lecomte, T., Pfeil, M., Green, O., Basagiannis, S., Sadovykh, A.: Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project. In: 2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data), pp. 1–6 (2016). DOI 10.1109/CPSData.2016.7496424
22. Masci, P., Zhang, Y., Jones, P.L., Oladimeji, P., D'Urso, E., Bernardeschi, C., Curzon, P., Thimbleby, H.: Combining PVSio with Stateflow. In: NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings, pp. 209–214 (2014). DOI 10.1007/978-3-319-06200-6_16
23. Mauro, G., Thimbleby, H., Domenici, A., Bernardeschi, C.: Extending a user interface prototyping tool with automatic MISRA C code generation. In: C. Dubois, P. Masci, D. Méry (eds.) Proceedings of the Third Workshop on Formal Integrated Development Environment, Limassol, Cyprus, November 8, 2016, *Electronic Proceedings in Theoretical Computer Science*, vol. 240, pp. 53–66. Open Publishing Association (2017). DOI 10.4204/EPTCS.240.4
24. Meadows, C.: Formal methods for cryptographic protocol analysis: emerging issues and trends. IEEE Journal on Selected Areas in Communications **21**(1), 44–54 (2003). DOI 10.1109/JSAC.2002.806125
25. Mercaldo, F., Nardone, V., Santone, A., Visaggio, C.: Hey malware, i can find you! pp. 261–262 (2016). DOI 10.1109/WETICE.2016.67
26. Mitchell, R., Chen, I.: Modeling and analysis of attacks and counter defense mechanisms for cyber physical systems. IEEE Transactions on Reliability **65**(1), 350–358 (2016). DOI 10.1109/TR.2015.2406860
27. Mitsch, S., Ghorbal, K., Platzer, A.: On provably safe obstacle avoidance for autonomous robotic ground vehicles. In: P. Newman, D. Fox, D. Hsu (eds.) Robotics: Science and Systems (2013)
28. Modelio web site (2018). `http://www.modelio.org` retrieved 11/29/2018
29. Muñoz, C.: Rapid prototyping in PVS. Tech. Rep. NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA (2003)
30. Oladimeji, P., Masci, P., Curzon, P., Thimbleby, H.: PVSio-web: a tool for rapid prototyping device user interfaces in PVS. In: FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems, London, UK, June 24, 2013 (2013). DOI 10.14279/tuj.eceasst.69.963.944
31. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: D. Kapur (ed.) Automated Deduction — CADE-11, *Lecture Notes in Computer Science*, vol. 607, pp. 748–752. Springer Berlin Heidelberg (1992). DOI 10.1007/3-540-55602-8_217
32. Owre, S., Rushby, J., Shankar, N., Von Henke, F.: Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. IEEE Transactions on Software Engineering **21**(2), 107–125 (1995)
33. Palmieri, M., Bernardeschi, C., Masci, P.: Co-simulation of semi-autonomous systems: The line follower robot case study. In: Software Engineering and Formal Methods — SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers, pp. 423–437 (2017). DOI 10.1007/978-3-319-74781-1_29
34. Platzer, A., Quesel, J.D.: Keymaera: A hybrid theorem prover for hybrid systems. In: 3rd International Joint Conference on Automated Reasoning (IJCAR), vol. Lecture Notes in Computer Science, pp. 171–178 (2008). DOI 10.1109/ISRCS.2012.6309293
35. Yampolskiy, M., Horvath, P., Koutsoukos, X.D., Xue, Y., Sztipanovits, J.: Systematic analysis of cyber-attacks on cps-evaluating applicability of dfd-based approach. In: 2012 5th International Symposium on Resilient Control Systems, pp. 55–62 (2012). DOI 10.1109/ISRCS.2012.6309293