Modeling and Simulation of Attacks on Cyber-Physical Systems

Cinzia Bernardeschi, Andrea Domenici, Maurizio Palmieri

Department of Information Engineering
University of Pisa

Work presented at 3rd International Workshop on FORmal methods for Security Engineering - ForSE 2019
1. Overview of the contribution

2. Background
   Co-Simulation of Cyber-Physical Systems
   Prototype Verification System (PVS)

3. Modeling Attacks in PVS
   Formal modeling of attacks
   Modification to existing model

4. Co-simulation scenario
   Line Follower Robot

5. Conclusions
Contribution

The contribution consists in a case study that shows how a co-simulation environment can be used for modelling and simulate the effects of attacks in CPS.

Preliminary work:
- Manual control
- Attack to sensors and actuators

The co-simulation environment allows:
- The analysis of the behavior of the CPS under attack
- Formal proofs for CPS
Cyber Physical Systems (CPS)

CPS systems are characterized by the coexistence of continuous and discrete behaviors, and of heterogeneous subsystems.

- **discrete systems**: controller
- **continuous systems**: plant (differential equations)

Model-based development of CPSs can benefit from the availability of different modeling languages and tools, each tailored to different components or aspects.
Co-simulation of Cyber-Physical Systems

FMI: emerging standard for co-simulation of cyber-physical systems.

Standard interface for coupling of tools.
Master-Slave approach.
Functional Mock-up Unit (FMU)
Data exchange is restricted to discrete points.

FMI co-simulation with the INTO-CPS tool chain


INTO-CPS project case study: LFR

http://into-cps.org/


Industry follower groups

Case study: Line follower robot

cosimulation of a robot that can follow a line painted on the ground

the line contrasts from the background and the robot uses a number of sensors to detect light and dark areas on the ground.
PVSSio-web

A formal methods toolkit for model-based development of human-machine interfaces

Technologies involved

Ubuntu-Linux OS

- Prototype Verification System
  https://github.com/SRI-CSL/PVS
- PVSio
  https://github.com/pvsio-web
- Libwebsocket library
  https://libwebsockets.org/

PVS Theorem prover with an extensive number of inference rules
Providing an assisted formal verification process based on the Sequent Calculus.

The Prototype Verification System (PVS) is an interactive theorem prover developed at SRI International by S. Owre, N. Shankar, J. Rushby and others.

The PVS language builds theories based on classical typed higher-order logic (a pure declarative language)

PVSio (Ground evaluator) For each function a Lisp procedure to compute its value is generated.

PVSio-web is an open-source environment enabling developers and users of interactive devices to assess and validate them with respect to human-machine interaction. Implemented in JavaScript by a software platform composed of several scripts, invoked through a web interface.
Semi-autonomous Line Follower Robot

Graphical interface
Javascritp - HTML5

WebSocket client

Commands related to user actions on device buttons

State of the device after the evaluation of the commands

WebSocket server

PVS executable model/Simulator

FMI interface

Web browser

Manual control

Semi-autonomous Line Follower Robot

Sensor Left (20 sim)
Sensor Right (20 sim)
Controller (PVSio-web)
Robot (20 sim)

CO-SIMULATION MASTER
Semi-autonomous Line Follower Robot

LFR: robotic model

Manual operating mode
the operator can send commands to the robot using a gamepad controller.

Automatic operating mode
the robot behaves like in the original INTO-CPS example.

original robot model is extended to support two operating modes

the console display reports robot status information, such as robot position, and speed
Semi-autonomous Line Follower Robot

State: TYPE =

[#lightSensors: LightSensors,
motorSpeed: MotorSpeed,
gear: Gear,
cc: CruiseControl#]

init.state: State =

(#lightSensors := (#left := 0, right := 0#),
motorSpeed := (#left := IDLE, right := IDLE#),
gear := DRIVE,
cc := AUTO#)

BRAKE_STEP: Speed = \frac{1}{10}

brake(st: State): State =

st  

WITH [cc := MANUAL,
motorSpeed := (#left := dec.speed(motorSpeed(st)‘left, BRAKE_STEP),
right := dec.speed(motorSpeed(st)‘right,
BRAKE_STEP)#]
Considerations on the attacks

• Attack to sensors
  • Can be masked (if fault tolerance techniques are applied) attack --- malicious fault
  • Can alter the value of local/hidden variables of the control
  • Can affect logging/tracing mechanism of the system

• Attack to actuators
  • Can have more impact on the behaviour of the system

A combination of them.
Considerations on the attacks

The attacker can acquire control of the robot from the joystick, manually control the robot with buttons, and change its speed or direction.
Considerations on the attacks

Attacks generated internally by the simulation algorithm

Attacks to sensors
• The effect is the corruption of data received from sensors
• The output written at the end is based on corrupted inputs

Attacks to actuators
• The effect is the corruption of data sent to actuators
• The output written at the end ignores the computed one

Implementation: attacks as functions that alter the state of the controller.
Introducing Attacks in control components

The attack is triggered only if certain condition is met

- Always from the beginning
- Always after a specific timestep
- Always after a random timestep
- During random intervals
- During specific intervals
- Only on specific timesteps
- Only on random timesteps
- Only once

Different models of the attack injector

Injector specified as a timed automaton (TA)

\[ [t == k] \leftarrow \text{sens}_V ! t := 0 \]


Formal modeling of attacks on CPS

\[ A = \langle \text{Var}_A, \text{Clk}_A \cup \{\text{stepCounter}\}, \text{Com}_A \rangle \]

• \( \text{Var}_A \) is the set of variables altered by the attack \( A \)
• Two types of clocks are used:
  • \( \text{Clk}_A \) represents the set of attacker clocks
  • \( \text{stepCounter} \) represents the global time
• \( \text{Com}_A \) is a set of guarded statements in the form:
  • \( \text{Condition} \rightarrow x_1 := v_1; \ldots; x_n := v_n \)
• \( \text{Condition} \) is a guard on clocks
• \( x \in \text{Var}_A \cup \text{Clk}_A \)
PVS skeleton of attack function

attack(st: ext_State): ext_State =

IF condition
    THEN st
        WITH [ 
            x1 := v1,
            ...
            xn := vn
        ]
    ELSE st
ENDIF

ext_state :
the state of the controller + $Clk_A + stepCounter$

The IF-THEN-ELSE statement represents $Com_A$

$x1, ..., xn \in Var_A \cup Clk_A$

If no condition is met the state of the system is not modified
Sensor/Actuator attacks in PVS

```plaintext
model_under_attack(st: ext_State) : ext_State =
   LET st1 = Sensor_attack(st),
   IN tick(st1)
```

```
tick(st: ext_State): ext_State
computes the output and also increments stepcounter.
```

```plaintext
model_under_attack(st: ext_State) : ext_State =
   LET st1 = tick(st),
   IN Actuator_attack(st1)
```

Attack on sensors

```
Control loop(){
   readInput();
   model_under_attack();
   writeOutput();
}
```

Attack on actuators

MeTRiD 2019 – April 6, 2019
Example: Line Following Robot (LFR)

State: TYPE = [#
lightSensors: LightSensors,
motorSpeed: MotorSpeed,
time: real,
cc: CruiseControl #]

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

ext_State: TYPE = [#
state: State,
stepcounter: int,
clk1: int,
clk2: int,
clk3: int #]

tick(st:ext_State): ext_State = st WITH[
  state := tick(st.state),
  stepcounter := stepcounter +1]
The attack sporadically switches off each wheel for the duration of a single step.

```
Actuator_attack(st: ext_State): ext_State =
IF clk2(st) = clk3(st) THEN
  st WITH [
    state`motorSpeed := (#
      left := 0,
      right := 0 #),
    clk2 := NRANDOM(20),
    clk3 := 0 ]
ELSE st WITH [ clk3 := clk3 + 1 ]
ENDIF
```

- \( V_{\text{ar}}_{\text{Actuator\_attack}} = \{\text{motorSpeed}\} \)
- \( C_{\text{lk}}_{\text{Actuator\_attack}} = \{\text{clk}2, \text{clk}3\} \)
- \( C_{\text{om}}_{\text{Actuator\_attack}} = \{\text{Actuator\_attack}\} \)

Clk2 is initialized with a random value
Clk3 is initialized with zero
The attack forces the value of the left sensor starting from a random timestep.

```
Sensor_attack(st: ext_State): ext_State =
  IF stepCounter(st) >= clk1(st)
    THEN st WITH [
      state`lightSensors`left := 140]
  ELSE st
ENDIF
```

\[
\begin{align*}
\text{Var}_{\text{Sensor_attack}} &= \{\text{lightSensor}'\text{left}\} \\
\text{Clk}_{\text{Sensor_attack}} &= \{\text{clk1}\} \\
\text{Com}_{\text{Sensor_attack}} &= \{\text{Sensor_attack}\}
\end{align*}
\]

Clk1 is initialized with a random value.
Value below the threshold of 150 means “black”
Co-simulation: no attack

- Using the INTO-CPS Application, we have run different co-simulation scenarios
  - fixed stepsize of 0.01 seconds
  - duration of 20 seconds.

- The first scenario is the LFR without any attack
Co-simulation: attack on actuators

- The second scenario is the one where we applied the attack on the actuator
  - The robot behaves in a consistent way, but it has encountered a reduction of performance
Co-simulation: attack on sensors

- The third scenario is the one where we applied the attack on the sensor
  - The robot is forever stuck in a circle
Automatic FMU generation from a graphic PVSio-web editor
Formal verification is an important complement to co-simulation

For example, we can prove that the LFR system satisfies this property:

It is always the case that if the value of the right sensor is set to white, the power of the left motor is lower than or equal to the power of the right motor.

never_turn_right: THEOREM
kth_step(NRANDOM(500)+1)`lightSensors`right>150 IMPLIES
FORALL ((K:above(NRANDOM(500)+1)) :
    motorSpeed(kth_step(K))`left <= -motorSpeed(kth_step(K))`right
Summary

• We allowed the study of attacks on CPS with formal methods and co-simulation

• We have provided an easy way to include attacks in PVS model
  • Encouraging the use of formal methods in model-based design

• The extended models can be simulated together with models from different tools
  • Enhancing the validation of the model
  • Reducing the effort required to formally specify a Cyber-Physical System
  • Exploiting the advantages offered by well known tool-chains
Conclusions

• To assess safety properties of the system under attack
  • By exploiting the PVS theorem prover

• To allow end-user training in recognizing attack scenarios
  • By implementing interactive attacks with PVSio-web

• To find critical elements of the system and provide mitigation mechanism
  • By exploiting the results of the previous points