REAFFIRM: A Framework for Model-Based Repair of Resilient Cyber-Physical Systems

Luan Viet Nguyen, Gautam Mohan, James Weimer, Oleg Sokolsky, Insup Lee, Rajeev Alur
Department of Computer and Information Science
University of Pennsylvania
{luanvn, gmohan1, weimerj, sokolsky, lee, alur}@seas.upenn.edu

ABSTRACT

Model-based design offers a promising approach for assisting developers to build a reliable and secure cyber-physical systems (CPSs) in a systematic manner. In this methodologies, a designer first constructs a model, with mathematically precise semantics, of the system under design, and performs extensive analysis with respect to correctness requirements before generating the implementation from the model. However, as new vulnerabilities are discovered, requirements evolve aimed at ensuring resiliency. Current methodology demands an expensive, and at times infeasible, redesign and reimplementation of the system from scratch. In this paper, we propose a new methodology, which we call REAFFIRM to facilitate the integration of evolving resiliency requirements in model-based design and verification for CPSs. REAFFIRM contains two main modules, a model transformation, and a model synthesizer. The first module applies a given resilience pattern specified as a model transformation script to a partial behavior model and produces a candidate resilient model that contains parameters for which the values need to be synthesized to ensure that the safety requirement is satisfied. The second module takes a parametrized model as an input and performs the synthesis of parameter values that makes the requirement, expressed as a formula in the Signal Temporal Logic, satisfied. It internally uses the falsification tool Breach properly to search over the space of parameter values and find values for which the input safety property is violated. Based on the counterexample produced by Breach, parameter ranges are tightened and the process repeats until the property is satisfied. To evaluate the proposed approach, we use REAFFIRM to automatically synthesize resilient models for an adaptive cruise control (ACC) system under GPS sensor spoofing attacks, and for a single machine infinite bus (SMIB) system under sliding-mode switching attacks.

ACM Reference format:
Luan Viet Nguyen, Gautam Mohan, James Weimer, Oleg Sokolsky, Insup Lee, Rajeev Alur
. 2019. REAFFIRM: A Framework for Model-Based Repair of Resilient Cyber-Physical Systems. In *Proceedings of ICCPS Conference, Montreal, Canada, April 2019 (Conference'19),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

Conference'19, Montreal, Canada

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A cyber-physical system (CPS) consists of computing devices communicating with one another and interacting with the physical world via sensors and actuators. Increasingly, such systems are everywhere, from smart buildings to autonomous vehicles to mission-critical military systems. The rapidly expanding field of CPSs precipitated a corresponding growth in security concerns for these systems. The increasing amount of software, communication channels, sensors and actuators embedded in modern CPSs make them likely to be more vulnerable to both cyber-based and physical-based attacks [4, 19, 25, 43, 44]. As an example, *sensor spoofing* attacks to CPSs become prominent, where a hacker can arbitrarily manipulate the sensor measurements to compromise secure information or to drive the system toward unsafe behaviors. Such attacks have successfully disputed the braking function of the anti-lock braking systems [4, 38], and compromised the insulin delivery service of a diabetes therapy system [28]. Alternatively, attackers can gain access to communication channels to either manipulate the switching behavior of a smart power grid [31] or disable the brake system of a modern vehicle [26]. Therefore, it is essential for a designer to build a CPS model with the resilient capability to cope with different attack scenarios. However, many CPSs are developed today without considering resiliency at all. Model-based systems engineers often neglect or incompletely consider incorporating resilient patterns in their designs. Generally, constructing a behavioral model at design time that offers resiliency for all kinds of attacks and failures is notoriously difficult as it requires investigating many sub-components in an integrated manner, especially for a large-scale CPS.

Traditionally a model of a CPS consists of block diagrams describing the system architecture and a combination of state machines and differential equations describing the system dynamics [5]. Suppose the designer has initially constructed a model of a CPS that satisfies correctness requirements, but at a later stage, this correctness guarantee is invalidated, possibly due to the emergence of new requirements, or adversarial attacks on sensors, or violation of environment assumptions. There is currently lack of an automated mechanism that can efficiently repair the initial design and provide resilience. Many significant research have been introduced to build resilient CPSs such as the approach proposed in [17] that can be used to design a resilient CPS through co-simulation of discrete-event models, a modeling and simulation integration platform for secure and resilient CPS based on attacker-defender games proposed in [27] with the corresponding testbed introduced in [35], and the resilience profiling of CPSs presented in [21]. Although these approaches can leverage the modeling and testing for a resilient CPS, they do not offer a model repair mechanism as well as a generic approach to design a resilient pattern when vulnerabilities

are discovered. In most of the case, a designer needs to rebuild the system from scratch, which requires a lot of time and efforts.

In this paper, we propose a new methodology and an associated toolkit, which we call REAFFIRM, to assist a designer in repairing the partial behavior model to generate the completed behavior model with resilience. The proposed technique is relied on designing a collection of *potential edits* (or *resilient patterns*) to the original model to solve the *model synthesis problem*. For example, a conservative way ensuring safety upon encountering an unexpected or hazardous situation is to hand over control to a baseline safety controller. REAFFIRM takes the original design and known resilient patterns as inputs and then synthesizes the completed model that satisfies the system requirements. The resilient version generated by REAFFIRM may have additional modes of operation as well as new transitions between different modes, and as a result, has resilient behaviors derived from a counterexample returned by the *falsification* tool embedded within REAFFIRM. Since a counterexample characterizes undesirable, or *shall not*, situations, it can naturally describe how a system should respond when a particular sensor fails, or a previously unanticipated attack is identified. The model synthesizer within REAFFIRM can automatically integrate such a counterexample with state-machine based models thus allowing an incremental design to support resiliency. We also provide a new *model transformation language* for hybrid systems, called HATL. Such a transformation language allows a designer to specify a resilient pattern in a generic manner, without knowing the internal structures of a system. To the best of our knowledge, this transformation language is the first effort to design a programmable pattern for the modeling and repairing of resilient CPSs.

Overall, the main contributions of the paper are summarized as follows.

(1) the methodology to facilitate the integration of evolving resiliency requirements in model-based design and analysis for CPSs,
(2) the end-to-end design and implementation of the tool-chain, which integrates the model transformation and the model synthesis tools to automatically repair CPS models,
(3) the design and implementation of the model transformation language for specifying resilient patterns used to repair CPS models,
(4) the applicability of our approach on two proof-of-concept case studies where the resilient CPS models can be automatically constructed to mitigate practical attacks.

We anticipate that our methodology proposed along with REAFFIRM for automatically conducting the model repair and specifying the patterns of resiliency will be contributions of significant interest to the research community in the design of resilient CPSs. The remainder of the paper is organized as follows. Section 2 presents an overview of our proposed methodology through the running example of an adaptive control system (ACC), and introduces the architecture of REAFFIRM. Section 3 describes our model transformation language used to design a resilient pattern for hybrid systems. Section 4 presents the model synthesizer of REAFFIRM. Section 5 presents two proof-of-concept case studies that illustrates the capability of REAFFIRM in automatically repairing the original models of a) the ACC system under sensor spoofing attacks and b) the smart power grid system under sliding-mode attacks. Section 6 reviews the related works to REAFFIRM. Section 7 concludes the paper and presents future research directions for the proposed work.

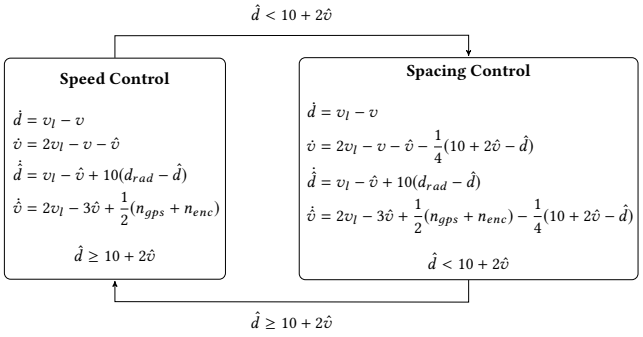

Figure 1: Original ACC model

2 OVERVIEWS OF METHODOLOGY

In this section, we will explain our methodology through the running example of an adaptive control system (ACC). This system has been designed to satisfy safety requirements pertaining to vehicle spacing that varies with vehicle speed. Assume that a designer has previously modeled an ACC system as a combination of the vehicle dynamics and an control module, and resiliency was not considered in the initial design. In the following, we will describe the ACC system as originally designed, its attack scenario, and an example of resilient pattern. Then, we present the structure of REAFFIRM and demonstrate how REAFFIRM can automatically perform a model transformation and synthesis to construct a new ACC model with resiliency.

2.1 Original ACC Model

The original ACC system operates in two modes: *speed control* and *spacing control*. In speed control, the host car travels at a driver-set speed. In spacing control, the host car aims to maintain a safe distance from the lead car. The ACC system decides which mode to use based on the real-time sensor measurements. For example, if the lead car is too close, the ACC system switches from speed control to spacing control. Similarly, if the lead car is further away, the ACC system switches from spacing control to speed control. In other words, the ACC system makes the host car travel at a driver-set speed as long as a safe distance is maintained.

The vehicle has two states in which d is the distance to the lead car, and v is the speed of the host vehicle. These states evolve according to $\dot{d} = v_l - v$ and $\dot{v} = -v + u$, where v_l is the speed of the lead vehicle and u is the control input. In this example, we assume that the lead vehicle speed is known exactly (e.g., it is communicated between vehicles). Conceptually, the dynamics for d represent that the derivative of the relative distance is the difference between the lead vehicle speed and the host vehicle speed. The dynamics for the host vehicle speed indicate that as the vehicle speed increases, it takes more acceleration (i.e., force) provided by the controller (i.e., engine) to maintain speed.

An ACC equipped vehicle has sensors that measure its velocity v via noisy wheel encoders, $v_{enc} = v + n_{enc}$, and a noisy GPS sensor, $v_{gps} = v + n_{gps}$, where n_{enc} and n_{gps} denote the encoder and GPS noisy, respectively. Additionally, the ACC system has a radar sensor that measures the distance to a (potential) lead vehicle, $d_{rad} = d + n_{rad}$. To design a control law, we need to estimate the state of the vehicle (i.e., we need an estimate of d and v, which we will call $\hat{d}$ and $\hat{v}$). To estimate the distance and velocity, we employ state estimators

$$\dot{\hat{d}} = v_l - \hat{v} + 10(d_{rad} - \hat{d})$$
$$\dot{\hat{v}} = -\hat{v} + u + \frac{1}{2}((v_{gps} + v_{enc}) - \hat{v})$$

To implement a controller, a control law based on the state estimates in speed and distance control modes are given:

$$u_s = v_l - (\hat{v} - v_l)$$
$$u_d = v_l - (\hat{v} - v_l) - \frac{1}{4}(d_{ref} - \hat{d})$$

where u_s is the control law in the speed mode, u_d is the control law in the spacing mode, and $d_{ref} = 10 + 2\hat{v}$. These control laws incorporate a reference velocity v_l, which can be thought of as constant gain that depends on the lead (or desired) vehicle velocity, while the other terms depend on the deviation of the lead vehicle and and ego vehicle states. Switching between modes is handled by monitoring the state estimates. The designer models the complete ACC system (including vehicle dynamics) as a hybrid system, illustrated in Figure 1. Here, the transition from speed control to spacing control occurs when the estimate of the distance is less than twice the estimated safe distance, i.e., $\hat{d} < 10 + 2\hat{v}$. A similar condition is provided for transitioning from spacing control to speed control, i.e., $\hat{d} \geq 10 + 2\hat{v}$.

2.2 Safety Violation under GPS Sensor Attack

In this example, the functional safety specification of the ACC system is specified that d should always be greater than $d_{safe} = 5 + v$. Assume that the designer has verified the ACC system safety requirement under the scenario when $d(0) \geq 20$, $v(0) \leq 30$, $|d(0) - \hat{d}(0)| \leq 1$, $|v(0) - \hat{v}(0)| \leq 1$, $v_l \geq 0$, $|n_{enc}| \leq 0.05$ and $|n_{gps}| \leq 0.05$. After designing the initial ACC system, it is determined that the GPS sensor can be *spoofed* [24, 41]. GPS spoofing occurs when incorrect GPS packets (possibly sent by a malicious attacker) are received by the GPS receiver. In the ACC system, this allows an attacker to arbitrarily change the GPS velocity measurement. Thus, a new scenario occurs when the assumption that $|n_{gps}| \leq 0.05$ is omitted, and the new assumption is $|n_{gps}| \leq \infty$. As a result, the safety specification could be violated under the GPS sensor attacks, and a designer needs to repair the original model with a resilient pattern.

2.3 Example of Resilient Pattern

To provide resilience against the GPS attacks, a potential strategy is to ignore the GPS value in resilient modes and use only the wheel encoders to estimate velocity. Since the ACC system has redundancy in the sensory information of its estimated velocity, the model synthesizer can repair the model by replacing the GPS velocity measurement with the wheel encoder velocity measurement when the GPS measurement significantly deviates from the wheel encoder measurement. Thus, the proposed fix then is first to create a resilient copy of the original model where the controller simply ignores the GPS reading as it can no longer be trusted. Then, adding new transitions from the legacy speed and spacing modes of the original model to the new resilient speed and spacing modes of the copy that uses only the wheel encoder as a velocity measurement source. We note that this transformation is generic, that is, it can be applied in a uniform manner to any given model simply by creating a duplicate version of each original mode and transition, copying the dynamics in each mode, but without a reference to the variable n_{gps}.

```
model = getModelByName("ACC") # retrieve the ACC model
model_copy = copyModel(model) # make a model copy

# start a transformation
addParam(model,"theta") # add new parameter theta

formode m = model.Mode {
    m_copy = addState(model,m)
    replace(m_copy.flow,"ngps","nenc")
    addTransition(model,m,m_copy,"abs(ngps-nenc)>theta")
}

fortran t = model_copy.Trans {
    # get source and destination modes of transition t
    src = t.source
    dst = t.destination
    # retrieve copies of source and destination modes
    src_copy = getCopyState(model,src)
    dst_copy = getCopyState(model,dst)
    addTransition(model,src_copy,dst_copy,t.guard)
}
# end of the transformation
```

Figure 2: An example of a resilient pattern written as a HATL script for the ACC system.

Figure 2 shows an example of a resilient pattern written as a transformation script that specifies a transformation from the original ACC model shown in Figure 1 to the parametrized model that enables resilience. In this script, we first create a copy of the original model. Next, we iterate over each mode of the model by calling the *formode loop*, make a copy with replacing the variable n_{gps} by n_{enc}, and then add a new transition from the original mode to the copied mode with a new guard condition. This guard condition is a constraint specified over the difference between n_{gps} by n_{enc} and a new parameter θ, which is added into the model using a function call *addParam*. Observe that while it would be possible to use only the wheel encoder all the time, a better velocity estimate must be obtained by using an average velocity measurement (from both the GPS and wheel encoders) when the GPS sensor is performing within nominal specifications. The main analysis question is when should the model switch from original copy to this resilient copy. In this example, we model this switching condition as $|n_{gps} - n_{enc}| \geq \theta$, where θ is an unknown parameter.

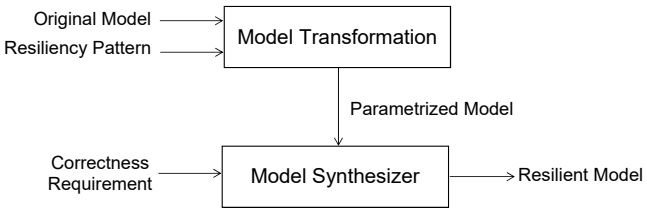

Figure 3: REAFFIRM Overview.

2.4 REAFFIRM Toolkit

Our REAFFIRM prototype for the model-based design and repair with resiliency consists of two tools, corresponding to a *model transformation* and a *model synthesizer* as shown in Figure 3.

- **Model Transformation** takes a given initial model and a resiliency pattern, and generates a partial model that contains *holes*, that is, parameters for which values need to be determined to ensure that the correctness requirements are satisfied. The resiliency pattern captures a generic way of transforming models that corresponds to commonly known mitigation strategies. The parameters in the incomplete model correspond to unknown switching conditions, or unknown assignments in variable updates, or even unknown coefficients in controller dynamics. The model is specified using the MathWorks Simulink/Stateflow (SLSF) format, and the correctness requirement is specified in the temporal logic Signal Temporal Logic (STL) that is widely used in tools for verification for cyber-physical systems [32]. The resiliency pattern can be specified as a *model transformation script* that operates on the internal representation of SLSF models specifying the desired transformation in a generic way. As an example, for a system that contains a nominal controller and a safety controller, a resiliency pattern can be specified as a transition from nominal controller to safety controller.
- **Model Synthesizer** takes a parameterized model (SLSF) and a correctness requirement (STL formula) as inputs and outputs a completed model with parameter values instantiated so as to satisfy the correctness requirements. Internally, the tool utilizes an open-source model falsification tool—Breach [15], to check an SLSF model against the specification. The counterexamples returned by this falsification tool are used to determine values for the desired parameters. At the end of a falsification loop, the completed model produced by REAFFIRM, compared to the partial behavior model, can have additional modes of operation as well as new transitions between different modes, and as a result has resilient behaviors specified in the resilient patterns.

To provide a resilience for the ACC example, the transformation tool of REAFFIRM will take the original ACC model shown in Figure 1 and the transformation script shown in Figure 2, and then output the *parameterized* model shown in Figure 4, where the variable n_{gps} is replaced by the variable n_{enc} in the resilient speed and spacing control modes, and the value of θ needs to be determined. Next, the model synthesizer of REAFFIRM takes the parameterized ACC model, the safety requirement encoded as an

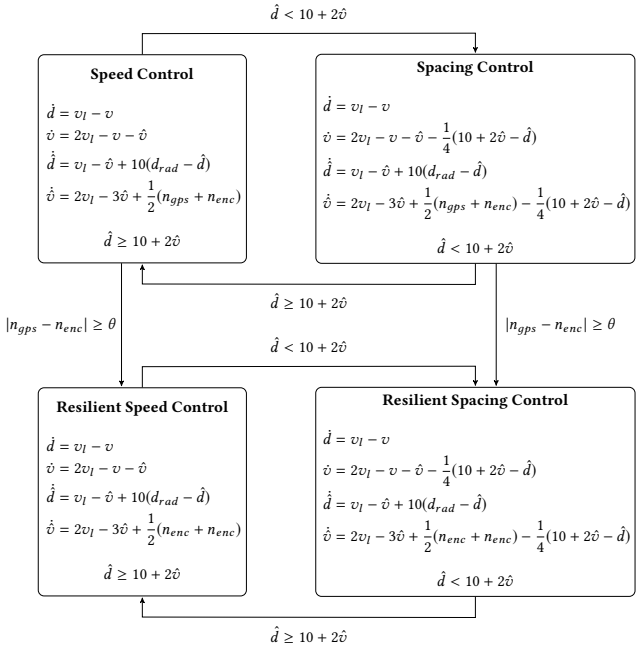

Figure 4: Updated Resilient ACC model

STL formula $\square_{[0,\infty]}(d[t] < 5 + v[t])$, and the specific range of θ, and then calls the synthesizer to determine the best θ which ensures the final model will always satisfy the safety requirement. If the tool cannot find any value of θ over a given range, it will suggest a designer to either search over a wider parameter range or try a different resilient pattern.

3 MODEL TRANSFORMATION

3.1 Representation of Hybrid System

Hybrid automata [5] are a modeling formalism popularly used to model hybrid systems which include both continuous dynamics and discrete state transitions. A hybrid automaton is essentially a finite state machine extended with a set of real-valued variables evolving continuously over intervals of real-time [5].[1] The main structure of a hybrid automaton $\mathcal{H}$ includes the following components.

- *Var*: the finite set of n continuous, real-valued variables; we have $Var = \mathcal{X} \cup \mathcal{P}$, where $\mathcal{X} \in \mathbb{R}^n$ is the set of n state variables and $\mathcal{P} \in \mathbb{R}^p$ is the set of p parameters. Moreover, *Var* is the disjoint of the set of input variables $\mathcal{I}$ and the set of output variables $\mathcal{O}$.
- *Mode*: the finite set of discrete modes. For each mode $m \in Mode$, $m.inv \subseteq \mathbb{R}^{n+p}$ denotes the invariant of mode m, and $m.flow \subseteq \mathbb{R}^n$ describes the continuous dynamics governed by a set of different equations. $\mathcal{Q} \triangleq Mode \times \mathbb{R}^n$ is the state space.
- *Trans*: the finite set of transitions between modes. Each transition is a tuple $\tau \triangleq \langle source, destination, guard, reset \rangle$, where

[1] In this paper, we only consider *deterministic* systems, where the system produces the same output for a given input. Note the contrast with *stochastic* systems in which one or more elements of the system have randomness associated with them; for example, the value of some system parameter may be extracted from a probability distribution. As a result, the stochastic system may yield different outputs for a given input.

source is a source mode and *destination* is a target mode that may be taken when a guard condition *guard* is satisfied, and the post-state is updated by an update map *reset*.

- *Init* is an initial condition, and $Init \subseteq Q$.

We use the dot (.) notation to refer to different components of tuples e.g., $\mathcal{H}.Trans$ refers to the transitions of automaton $\mathcal{H}$ and $\tau.guard$ refers to the guard of a transition τ. We note that the *model transformation language* proposed in this paper transforms a hybrid automaton based on modifying these components in a generic manner. The transformation tool of REAFFIRM can this transformation script and translate it into an equivalent script that performs a model transformation for different modeling framework of hybrid automata such as a continuous-time Stateflow chart.

Continuous-time Stateflow Chart. In this paper, we represent hybrid automata using *continuous-time* Stateflow chart, which is a standard commercial modeling language for hybrid systems integrated within MathWorks Simulink. Continuous-time Stateflow chart[2] supplies methods for engineers to quickly model as well as efficiently refine, test, and generate code for hybrid automata. The syntactic components of a continuous-time Stateflow chart are described similar to a hybrid automaton where a mode is a *state* associated with different types of actions including a) *entry* action executed when entering the state, b) *exit* executed when exiting the state, and c) *during* action demonstrates the continuous-time evolution of the variables (i.e., *flow* dynamics) when no transition is enabled. A variable can be specified as *parameter*, *input*, *output*, and *local variable*. A Stateflow diagram is deterministic since its transition is urgent and executed with priorities. Intuitively, a transition in a Stateflow chart is triggered as soon as the transition guard condition is satisfied, while a hybrid automaton can stay at the current mode as long as its invariant still holds. To overcome this gap, a recent work proposed in [10] provides an equivalent translation for both classes of deterministic and non-deterministic hybrid automata to Stateflow diagrams. Other significant research have been done to translate back and forth between hybrid automata and Simulink/Stateflow models [6, 33, 34].

3.2 Hybrid Automata Transformation Language

In our approach, the partial model of the system, which satisfies functional but not necessarily resiliency requirements is originally modeled in the form of hybrid automata. The model transformation that is at the core of REAFFIRM tool will then attempt to add resilience modules to the system and modify switching logic between modes of the automata, by applying resilience patterns. In order to specify resilient patterns for hybrid automata, we introduce a new transformation language called HATL (Hybrid Automata Transformation Language). The goal of HATL is to allow a designer to repair an original model in a programmable fashion. A script written in HATL is a sequence of *statements* that specify the changes over the structure of given hybrid automata. There are three types of statements in HATL including (a) the *loop* statement that iterates over the modes and transitions of a hybrid automaton,

[2]In this paper, we focus only on continuous-time Stateflow diagram that does not include hierarchical states.

```
root = sfroot;
diagram = root.find('-isa','Simulink.BlockDiagram');
model = model.find('-isa', 'Stateflow.Chart');
model_copy = copyModel(model);

addParam(model,"theta")

modes = getStates(model);
for i = 1 : length(modes)
  m = modes(i);
  m_copy = addState(model,m);
  m_copy.Label = strrep(m_copy.Label,"ngps","nenc");
  addTransition(model,m,m_copy,"abs(ngps-nenc)>theta");
end

trans = getTransitions(model_copy);
for i = 1 : length(trans)
  t = trans(i);
  if notDefaultTransition(t)
    src = t.Source;
    dst = t.Destination;
    src_copy = getCopyState(model,src)
    dst_copy = getCopyState(model,dst)
    addTransition(model,src_copy,dst_copy,t.LabelString)
  end
end
```

Figure 5: A translated MATLAB script for the HATL script shown in Figure 2.

(b) the *function call* statement that represents basic *transformation rule* implemented as in a standard library for HATL. A function call statement may or may not have arguments. Several examples of such a basic function call is *addMode*() (i.e., generating a new empty mode), or *addTransition*(m, m', g) (i.e., creating a transition from mode m to m' with a guard condition g). (c) The *assignment* statement sets a value (possibly returned by a function call) to a variable.

The model transformation tool built in REAFFIRM takes a resilient pattern written as HATL script, parses it to an intermediate representation, which is a set of data structures encoding the syntax of a hybrid automaton in Python. As we represent hybrid automata as continuous-time Stateflow charts, The HATL script then will be translated to a MATLAB script that performs a modification to the Stateflow chart through its API (Application Programming Interface). As an example, Figure 5 describes the translated MATLAB script of the HATL script shown in Figure 2. The translation to MATLAB is straightforward except that the iteration over each transition in MATLAB should neglect the *default transition*[3].

3.3 Implementation

4 MODEL SYNTHESIS

In this section, we present the model synthesizer incorporated in REAFFIRM which takes a parameterized model produced by the model transformation, and a correctness requirement as inputs, and then generates a completed model with parameter values instantiated so as to satisfy the correctness requirements. Since the structure of the completed model already determined after the model transformation, the model synthesis problem then reduces

[3]While Stateflow chart requires that the default transitions should be explicitly specified for exclusive (OR) states at every level of a hierarchy, other modeling frameworks of hybrid automata often describe it separately in a configuration file.

to the *parameter synthesis problem*. Given a safety specification φ, let $\mathcal{P}_s$ be a set of parameters of a transformed hybrid automaton $\tilde{\mathcal{H}}$, find the best instance values of $\mathcal{P}_s$ over its domain $\bar{\mathcal{P}}_s$ so that $\tilde{\mathcal{H}} \models \varphi$. For example, the transformation of the ACC model shown in Figure 4 introduces a new parameter θ whose value needed to be determined so that the completed model will satisfy the safety requirement with respect to the same initial condition of the state variables and parameters domains of the original model.

4.1 Overview of Breach

In our proposed methodology, we incorporated Breach into the model synthesizer of REAFFIRM as an analysis mechanism to perform the falsification and parameter synthesis for hybrid systems. Given a hybrid system modeled as a Simulink/Stateflow diagram, an STL specification represented the safety property, and specific parameter ranges, Breach [15] can perform an optimized search over parameter domains to find parameter values that cause the system violating the given STL specification. The parameter mining procedure is guided by the counterexample obtained from the falsification, and it terminates if there is no counterexample found by the falsifier or the maximum number of iterations specified by a user is reached. On the other hand, Breach can compute the sensitivity of execution traces to the initial conditions, which can be used to obtain completeness results by performing systematic simulations. Moreover, Breach provides an input generator for engineers to specify different testing input patterns such as step, pulse width, sinusoid, and ramp signals. This input generator is designed to be extensible, so users can write a specific input pattern to test their model against particular attack scenarios.

We note that although Breach cannot completely prove the system correctness, it can efficiently find bugs existing in the initial design of CPS that are too complex to be formally verified [23]. These bugs are essential for an engineer to specify resilient patterns to repair the model. Moreover, the general problem of verifying a CPS modeled as a hybrid system is proved to be *undecidable* [20]. Instead, the falsification algorithms embedded within Breach are scalable and work properly for black-box hybrid systems with different classes of dynamics. Thus, in practice, engineers prefer to use counterexamples obtained by a falsification tool to refine their design. Our prototype REAFFIRM utilizes the advantages of Simulink/Stateflow modeling framework and the falsification tool Breach to design a resilient pattern and perform the model synthesis for a resilient CPS.

4.2 Model Synthesis using Breach

Next, we describe how to use Breach to synthesize parameters values for the parametrized model returned from the model transformation tool. The parameter synthesis procedure include following steps.

(1) We specify the initial conditions of state variables and parameters, the set of parameters $\mathcal{P}_s$ that need to be mined, their certain ranges of values $\bar{\mathcal{P}}_s$, and the maximum time (or number of iterations) for the optimization solver of Breach.
(2) Next, we call the falsifier loop within Breach to search for a counterexample. For each iteration, if the counterexample is exposed, the unsafe values of $\mathcal{P}_s$ will be returned. Based on these values, the tool will automatically update the parameter domain $\bar{\mathcal{P}}_s$ to the new domain $\bar{\mathcal{P}}'_s \subset \bar{\mathcal{P}}_s$, and then continue the falsification loop.
(3) The process repeats until the property is satisfied which means that the falsifier cannot find the counterexample until the user-specified limit on the number of iterations for the solver is reached.
(4) As a result, the tool returns the best (and safe) values of $\mathcal{P}_s$, updates the parametrized model with these values, and then exports the completed model.

Monotonic Parameters. The search over the parameter space of the synthesis procedure can be significantly reduced if the satisfaction value of a given property is monotonic w.r.t to a parameter value. Intuitively, the satisfaction of the formula monotonically increases (respectively decreases) w.r.t to a parameter p that means the system is more likely to satisfy the formula if the value of p is increased (respectively decreased). In the case of monotonicity, the parameter space can be efficiently truncated to find the *tightest* parameter values such that a given formula is satisfied. In Breach, the check of monotonicity of a given formula w.r.t specific parameter is encoded as an STM query and then is determined using an STM solver. However, the result may be *undecidable* due to the undecidability of STL [22]. In this paper, the synthesis procedure is based on the assumption of satisfaction monotonicity. If the check of monotonicity is undecidable over a certain parameter range, a user can manually enforce the solver with decided monotonicity (increasing or decreasing) or perform a search over a different parameter range.

5 MODEL REPAIR FOR RESILIENCY

In this section, we demonstrate the capability of REAFFIRM to repair CPSs models under unanticipated attacks. We first revisit the ACC model and evaluate more resilient patterns that can be applied to repair the model under the GPS sensor attack. Second, we investigate a sliding-mode switching attack that causes instability for a smart grid system, and how REAFFIRM can be applied to automatically repair the model under this attack.

5.1 Adaptive Cruise Control System

GPS Sensor Attack Construction. To perform a spoofing attack on the GPS sensor of the ACC model, we continuously inject a false data to manipulate its measurement value. In this case, we omit the original assumption $|n_{gps}| \leq 0.05$, and employ the new assumption as $|n_{gps}| \leq 50$. Using the input generator in Breach, we specify the GPS spoofing attack as a standard input test signal such as a constant, ramp, step, sinusoid or random signal whose amplitude varies over the range of [-50, 50].

Additional Resilient Patterns for ACC model. Under the GPS spoofing attack, the original ACC model does not satisfy its safety requirement and a designer needs to apply a certain resilient pattern to repair the model. The first example of such a resilient pattern for the ACC model have been introduced in Section 2, which makes a resilient copy of the original model where the controller simple ignores the GPS reading as it can no longer be trusted. However, we need to determine the best switching condition from the original

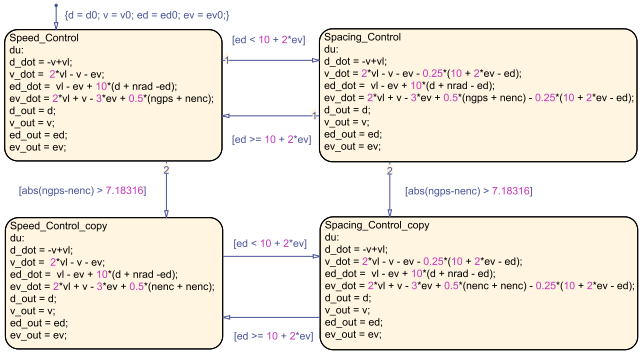

Figure 6: the resilient ACC model with a synthesized value of $\theta = 7.18316$.

model to the resilient copy. Figure 6 shows the completed, resilient model in which the switching condition is determined by synthesizing the value of θ using Breach. In the attack scenario, we assume that the GPS spoofing occurs at every time point, specified as a random constant signal over the range of [-50, 50].

The second resilient pattern is an extended version of the first one by specifying a switching-back condition from the resilient copy to the original model when the GPS sensor is detected and mitigated. An example of such a switching-back condition is when the difference between the n_{enc} and n_{gps} are getting smaller, i.e., $|n_{gps} - n_{enc}| < \theta - \epsilon$, where ϵ is a positive user-defined tolerant. For this pattern, the model transformation script can be written similar to the one shown in Figure 2 with adding the *addTransition* function from the copy mode to the original mode with the guard condition labeled as $|n_{gps} - n_{enc}| < \theta - \epsilon$ in the *formode* loop.

Alternatively, another resilient pattern, which we do not need to modify the structure of the original model, is to model the redundancy in the sensory information as a linear combination of different sensor measurements. For example, instead of taking the average of n_{gps} and n_{enc}, we can model their relationship as $\theta n_{gps} + (1 - \theta) n_{enc}$, and then synthesize the value of θ so that the safety property is satisfied. The transformation script of this resilient pattern is given in Figure 7. For this pattern, we assume that a designer still wants to use all sensor measurements even some of them are under spoofing attacks and would like to search for the value of θ over the range of [0.2, 0.8] (instead of [0, 1]). Given the same attack model for the other patterns, the synthesizer in REAFFIRM fails to find the value of θ within the given range to ensure the safety property holds. However, if we enlarge the range of θ to [0.1, 0.9], the synthesizer successfully find the safe value $\theta = 0.1543$, and the resilient model shown in Figure 8. This result indicates that the pattern can repair the model if the GPS spoofing attack specified over a small range, but will fail for a larger range (e.g., $|n_{enc}| \leq 100$).

5.2 Single-Machine Infinite-Bus System

Next, we study a class of cyber-physical switching attacks that can destabilize a smart grid system model, and then apply REAFFIRM to repair the model to provide resilience.

```
model = getModelByName("ACC") # retrieve the ACC model

# start a transformation
formode m = model.Mode {
    replace(m.flow,"ngps", "2*theta*ngps")
    replace(m.flow,"nenc", "2*(1-theta)*nenc")
}
# end of the transformation
```

Figure 7: The third resilient pattern for the ACC system based on the linear combination of n_{enc} and n_{gps}.

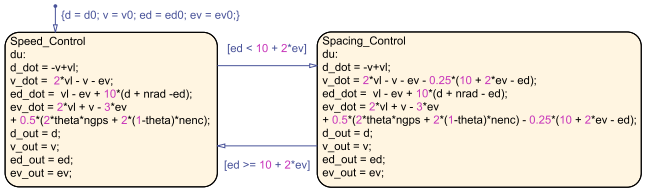

Figure 8: the resilient ACC model with a synthesized value of $\theta = 0.1543$ for the resilient pattern shown in Figure 7.

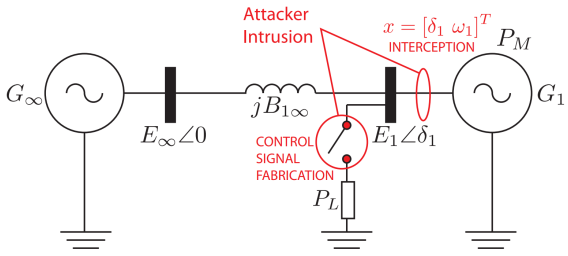

Figure 9: Single-machine infinite-bus model [16].

Original SMIB Model. A smart power grid system such as the Western Electricity Coordinating Council (WECC) 3-machine, 9-bus system [37], can be represented as a single-machine infinite-bus (SMIB) model shown in Figure 9[4]. In this model, G_∞ and G_1 correspondingly represent the SMIB and local generators; B_∞ and B_1 denote the infinite and local bus, respectively; E_∞ is the infinite bus voltage; E_1 is the internal voltage of G_1; $B_{1\infty}$ is the transfer susceptance of the line between B_1 and B_∞; and P_M is the mechanical power of G_1. The local load P_L is connected or disconnected to the grid by changing a circuit breaker status. The SMIB system is considered as a *switched system* in which the physical dynamics are changed between two operation modes based on the position of the circuit breaker. For an appropriate selection of parameters [16], the second-order swing equation which characterizes the transient stability of the local generator G_1 can be described as:

$$\dot{\delta_1} = \omega$$

$$\dot{\omega} = \begin{cases} -10 sin\delta_1 - \omega_1, & \text{if } P_L \text{ is connected.} \\ 9 - 10 sin\delta_1 - \omega_1, & \text{if } P_L \text{ is disconnected,} \end{cases}$$

where, δ_1, ω_1 are the deviation of the rotor angle and speed of G_1 respectively, and $x = [\delta_1, \omega_1]^T$ is the state vector of G_1.

[4]the figure is copied from [16].

The swing equation of the SMIB system shown above has an interesting property known as a *sliding mode* behavior. This behavior occurs when the state of the system is attracted and subsequently stays within the *sliding surface* defined by a state-dependent switching signal $s(x) \in \mathbb{R}$ [14, 30]. An example of a sliding surface is $s(x) = 0$. When the system is confined on a sliding mode surface, its dynamics exhibit high-frequency oscillations behaviors, so-called a *chattering* phenomenon, which is well-known in the power system design [36]. At this moment, if the attacker conducts the fast switchings between two operation modes, the system will be steered out of its desirable equilibrium position. As a result, the power system becomes unstable even if each individual subsystem is stable [30].

Sliding-mode Attack Construction. To successfully perform sliding-mode attack to a power grid system, we assume that the attack can a) gain some knowledge about the state information to mathematically construct an unstable sliding surface that can destabilize the system, and b) access to the communication channel to control the circuit breaker position. We note that a sliding-mode attack can be considered as a cyber-physical attack as it manipulate on both physical and cyber parts of the system. For the SMIB model with the swing equation defined previously, an attacker can use a sliding surface $s(x) = \delta_1 + \omega_1$ to calculate the value of the switching signal based on the following equation.

$$\dot{\delta_1} = \omega$$
$$\dot{\omega} = \begin{cases} -10sin\delta_1 - \omega_1, & s(x) \geq \epsilon. \\ 9 - 10sin\delta_1 - \omega_1, & s(x) < -\epsilon, \end{cases}$$

, where ϵ represents switching delays and hysteresis [31].

The stages of sliding-mode attack can summarized as follows.

(1) The attacker switches the circuit breaker to connect the load to the grid,
(2) when $\delta_1 + \omega_1 < -\epsilon$, the attacker switches the circuit breaker to disconnect the load,
(3) when $\delta_1 + \omega_1 \geq \epsilon$, the attacker switches the circuit breaker to connect the load,
(4) the attacker repeats steps (1) and (2) until the system is driven out of the stability boundary, and then
(5) permanently switches the circuit breaker to disconnect the load.

In this paper, we model the sliding-mode attack to the SMIB model as a Simulink/Stateflow diagram including two Stateflow models, where the plant model displayed in Figure 10 represents the physical dynamics of the system and the attack model is shown in Figure 11. In the plant model, δ_1 and ω_1 are represented by *delta* and *omega*, respectively; and the initial conditions are $delta0 \in [0, 1.1198]$ and $omega0 \in [0, 1]$. The discrete variable *load* captures the open and closed status of the circuit breaker. In the attack model, the attacker selects $\epsilon = 0.2$, and the local variable t captures a simulation duration. We note that two transitions from the first mode to the second mode are executed with priorities such that the load is permanently disconnected at some instance where $t \geq 2.5$ seconds. Figure 12 illustrates the examples of stable (i.e., without an attack) and unstable (i.e., a counterexample appearing under the sliding-model attack) behaviors of the SMIB system returned by running the falsifier of

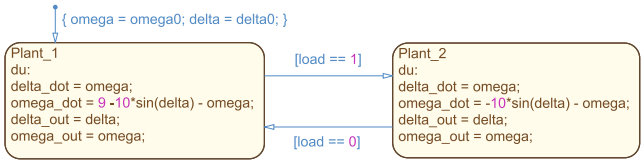

Figure 10: The Stateflow chart models the plant of the SMIB system.

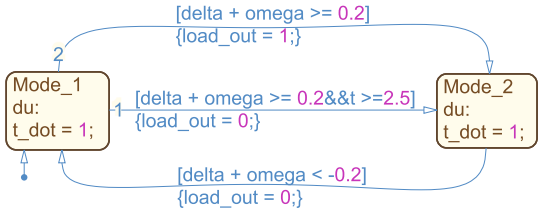

Figure 11: The Stateflow chart models the sliding-mode attack to the SMIB system.

Breach, respectively. The red box defines the stable (safe) operation region of the SMIB that can be formalized as the following STL formula,

$$\varphi_{smib} = \square_{[0,T]}(0 \leq \delta_1[t] \leq 3.5) \wedge (-2 \leq \omega_1[t] \leq 3), \quad (1)$$

where T is a simulation duration.

Resilient Pattern for SMIB Model. As a sliding-mode attack is constructed based on switching back-and-forth the circuit breaker quickly to trap the system inside the sliding surface before guiding its state variables evolving outside the stability boundary, a potential strategy to mitigate such an attack is to increase the minimum switching time of the circuit breakers. Indeed, the designer can repair the original model by including a minimum dwell time in each mode of the system to prevent rapid switching. Figure 13 shows a resilient pattern written as a HATL script that introduces the *clock* variable as a timer, and the switching time relies on the value of θ.

Resilient SMIB Model. Given a resilient pattern shown in Figure 13, the model transformation of REAFFIRM will convert the model to a new version that integrates the dwell-time pattern with the unknown parameter θ. Then, the model synthesis of REAFFIRM calls Breach to search for the best (i.e., minimum) value of θ over and the range of $[0, 1]$ that ensures the final model satisfies the STL Formula 1 under the sliding-mode attack. The final (resilient) model is shown in Figure 14, where the synthesized value of θ equals to 0.12. As a result, the resilient model is stable, and its simulation trajectories contain within a red box, similar to the most left subfigure shown in Figure 12.

6 RELATED WORKS

The idea of our methodologies proposed in this paper is inspired from the concept of *program sketching* [39, 40], and the recent works on completion of distributed protocols [7–9]. Given a model of the *incomplete* distributed protocol, which is a set of communicating finite-state processes with incomplete transition relations, a model

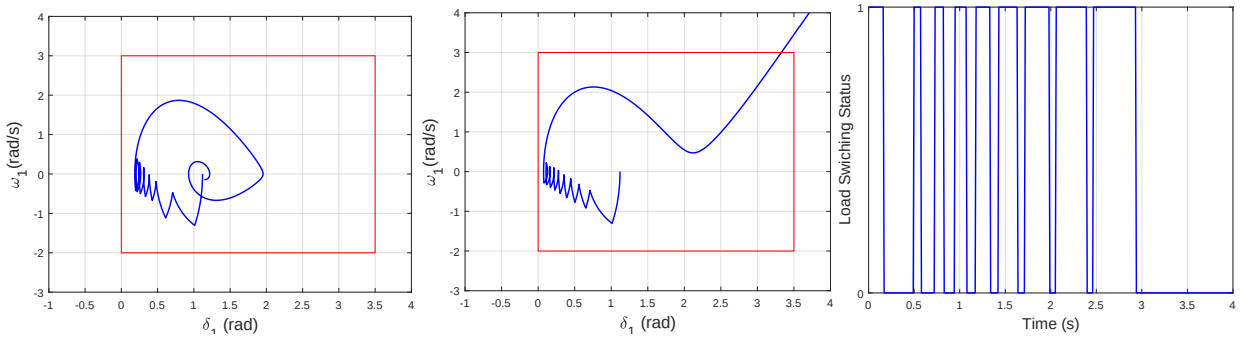

Figure 12: From left to right: 1) the stable system trajectory without an attack, 2) the counterexample represents the unstable system trajectory under the sliding-mode attack, and 3) the status of a circuit breaker during the attack, where 0 and 1 represent the disconnection and connection of the load P_L, respectively.

```
model = getModelByName("SMIB") # retrieve the plant model

# start a transformation
addParam(model,"theta") # add a new parameter theta
addLocalVar(model,"clock") # add a clock variable

formode m = model.Mode {
    addFlow(m,"clock_dot = 1")
}

fortran t = model.Trans {
    # a transition only triggers after theta seconds
    addGuardLabel(t,"&&","clock > theta");
    # reset a clock after each transition
    addResetLabel(t,"clock = 0");
}
# end of the transformation
```

Figure 13: A dwell-time resilient pattern for the SMIB system.

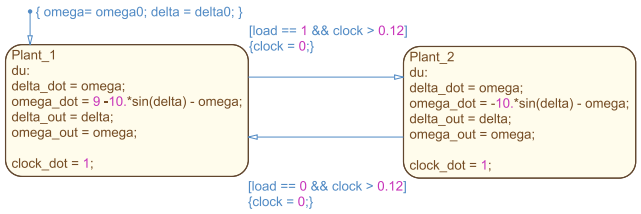

Figure 14: The resilient SMIB model with a synthesized dwell-time.

of the environment, and its correctness requirements specified in temporal logic [13], determine a *completion* of the finite state machines (FSMs) for the processes such that the composition satisfies all the requirements. This approach can be viewed as a fruitful collaboration between the designer and the synthesis tool where a programmer needs to provide a *skeleton* of the desired protocol, and some details that the programmer is unsure about, for instance, regarding corner cases and handling of unexpected messages, and the synthesizer will automatically complete it with adding transitions. Instead of synthesizing a *complete* FSM, we address the problem of automatic repair of CPSs modeled as hybrid automata, which is an extended FSM with a set of real-valued variables evolving continuously over intervals of real-time. In our approach, a designer needs to provide a model transformation script that transforms an initial model to a new version whose parameters values will be determined by the synthesizer to provide resilience.

In the context of the model transformation, GREAT is a metamodel-based graph transformation language that can be used to perform different transformations on domain-specific models [1, 2]. GREAT has been used to translate Simulink/Stateflow models to Hybrid Systems Interchange Format (HSIF) [3]. Such a translation scheme is accomplished by executing a sequence of translation rules described using UML Class Diagram in a specific order. Other approaches that also perform a translation from Simulink diagrams to hybrid systems formalisms such as Timed Interval Calculus [12], Hybrid Communicating Sequential Processes [29], Lustre [42], and SpaceEx [34]. HYST [11] is a conversion tool for hybrid automata which allows the same model to be analyzed simultaneously in several hybrid systems analysis tools. HYST takes a source input model in SpaceEx XML format [18], parses it to an intermediate representation, and then prints a resulting output to some desired formats. HYST can automatically translate hybrid automata to *trajectory-equivalent* Simulink/Stateflow models, which enables a *correct-by-construction* compositional design for CPS with embedding hybrid automata into large-scale SLSF models [10]. Unlike these approaches that focuses on the translation between different hybrid system modeling frameworks, our goal is to provide a light-weight programmable model transformation language for hybrid systems in which a designer only needs to write a simple script to repair the model.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented a new methodology and the tool-chain REAFFIRM that effectively assist a designer to repair CPS models under unanticipated attacks automatically. The model transformation tool takes a resilient pattern specified in the transformation language HATL and generates a new model including unknown parameters whose values can be determined by the synthesizer tool such that the safety requirement is satisfied. We demonstrated the high-applicability of REAFFIRM by using the tool-chain to efficiently repair CPS models under realistic attacks including the

ACC models under the GPS sensor spoofing attack and the SMIB models under the sliding-model attack.

Future Work

REFERENCES

[1] Aditya Agrawal, Gabor Karsai, and Ákos Lédeczi. 2003. An end-to-end domain-driven software development framework. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 8–15.

[2] Aditya Agrawal, Gabor Karsai, and Feng Shi. 2003. Graph transformations on domain-specific models. *Journal on Software and Systems Modeling* 37 (2003), 1–43.

[3] Aditya Agrawal, Gyula Simon, and Gabor Karsai. 2004. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science* 109 (2004), 43–56.

[4] Mohammad Al Faruque, Francesco Regazzoni, and Miroslav Pajic. 2015. Design methodologies for securing cyber-physical systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 30–36.

[5] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A Henzinger, P-H Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. 1995. The algorithmic analysis of hybrid systems. *Theoretical computer science* 138, 1 (1995), 3–34.

[6] Rajeev Alur, Aditya Kanade, S Ramesh, and KC Shashidhar. 2008. Symbolic analysis for improving simulation coverage of Simulink/Stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 89–98.

[7] Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2014. Synthesizing finite-state protocols from scenarios and requirements. In *Haifa Verification Conference*. Springer, 75–91.

[8] Rajeev Alur, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. 2015. Automatic completion of distributed protocols with symmetry. In *International Conference on Computer Aided Verification*. Springer, 395–412.

[9] Rajeev Alur and Stavros Tripakis. 2017. Automatic synthesis of distributed protocols. *ACM SIGACT News* 48, 1 (2017), 55–90.

[10] Stanley Bak, Omar Ali Beg, Sergiy Bogomolov, Taylor T Johnson, Luan Viet Nguyen, and Christian Schilling. 2017. Hybrid automata: from verification to implementation. *International Journal on Software Tools for Technology Transfer* (2017), 1–18.

[11] Stanley Bak, Sergiy Bogomolov, and Taylor T Johnson. 2015. HYST: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*. ACM, 128–133.

[12] Chunqing Chen, Jin Song Dong, and Jun Sun. 2009. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing* 21, 5 (2009), 451–483.

[13] Edmund M Clarke, Orna Grumberg, and Doron Peled. 2000. Model Checking. 2000. (2000).

[14] Raymond A DeCarlo, Stanislaw H Zak, and Gregory P Matthews. 1988. Variable structure control of nonlinear multivariable systems: a tutorial. *Proc. IEEE* 76, 3 (1988), 212–232.

[15] Alexandre Donzé. 2010. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*. Springer, 167–170.

[16] Abdallah K Farraj, Eman M Hammad, Deepa Kundur, and Karen L Butler-Purry. 2014. Practical limitations of sliding-mode switching attacks on smart grid systems. In *PES General Meeting— Conference & Exposition, 2014 IEEE*. IEEE, 1–5.

[17] John Fitzgerald, Ken Pierce, and Carl Gamble. 2012. A rigorous approach to the design of resilient cyber-physical systems through co-simulation. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. IEEE, 1–6.

[18] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*. Springer, 379–395.

[19] Thoshitha T Gamage, Bruce M McMillin, and Thomas P Roth. 2010. Enforcing information flow security properties in cyber-physical systems: A generalized framework based on compensation. In *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*. IEEE, 158–163.

[20] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. 1995. What's decidable about hybrid automata?. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. ACM, 373–382.

[21] Mark Jackson and J Fitzgerald. [n. d.]. Resilience profiling in the model-based design of cyber-physical systems. In *14th Overture Workshop: Towards Analytical Tool Chains, Technical Report ECE-TR-28*. 1–15.

[22] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V Deshmukh, and Sanjit A Seshia. 2015. Mining requirements from closed-loop control models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 34, 11 (2015), 1704–1717.

[23] James Kapinski, Jyotirmoy Deshmukh, Xiaoqing Jin, Hisahiro Ito, and Ken Butts. 2015. Simulation-guided approaches for verification of automotive powertrain control systems. In *American Control Conference (ACC), 2015*. IEEE, 4086–4095.

[24] Andrew J Kerns, Daniel P Shepard, Jahshan A Bhatti, and Todd E Humphreys. 2014. Unmanned aircraft capture and control via GPS spoofing. *Journal of Field Robotics* 31, 4 (2014), 617–636.

[25] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, and Srivaths Moderator-Ravi. 2004. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*. ACM, 753–760.

[26] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. 2010. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 447–462.

[27] Xenofon Koutsoukos, Gabor Karsai, Aron Laszka, Himanshu Neema, Bradley Potteiger, Peter Volgyesi, Yevgeniy Vorobeychik, and Janos Sztipanovits. 2018. SURE: A Modeling and Simulation Integration Platform for Evaluation of SecUre and REsilient Cyber–Physical Systems. *Proc. IEEE* 106, 1 (2018), 93–112.

[28] Chunxiao Li, Anand Raghunathan, and Niraj K Jha. 2011. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *e-Health Networking Applications and Services (Healthcom), 2011 13th IEEE International Conference on*. IEEE, 150–156.

[29] Jiang Liu, Jidong Lv, Zhao Quan, Naijun Zhan, Hengjun Zhao, Chaochen Zhou, and Liang Zou. 2010. A calculus for hybrid CSP. In *Asian Symposium on Programming Languages and Systems*. Springer, 1–15.

[30] Shan Liu, Bo Chen, Takis Zourntos, Deepa Kundur, and Karen Butler-Purry. 2014. A coordinated multi-switch attack for cascading failures in smart grid. *IEEE Transactions on Smart Grid* 5, 3 (2014), 1183–1195.

[31] Shan Liu, Xianyong Feng, Deepa Kundur, Takis Zourntos, and Karen Butler-Purry. 2011. A class of cyber-physical switching attacks for power system disruption. In *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*. ACM, 16.

[32] Oded Maler and Dejan Nickovic. 2004. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Springer, 152–166.

[33] Karthik Manamcheri, Sayan Mitra, Stanley Bak, and Marco Caccamo. 2011. A step towards verification and synthesis from simulink/stateflow models. In *Proceedings of the 14th international conference on Hybrid systems: computation and control*. ACM, 317–318.

[34] Stefano Minopoli and Goran Frehse. 2016. SL2SX translator: from Simulink to SpaceEx models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. ACM, 93–98.

[35] Himanshu Neema, Bradley Potteiger, Xenofon Koutsoukos, Gabor Karsai, Peter Volgyesi, and Janos Sztipanovits. 2018. Integrated simulation testbed for security and resilience of CPS. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. ACM, 368–374.

[36] Asif Sabanovic, Leonid M Fridman, Sarah Spurgeon, and Sarah K Spurgeon. 2004. *Variable structure systems: from principles to implementation*. Vol. 66. IET.

[37] Peter W Sauer and MA Pai. 1998. Power system dynamics and stability. *Urbana* (1998).

[38] Yasser Shoukry, Paul Martin, Paulo Tabuada, and Mani Srivastava. 2013. Non-invasive Spoofing Attacks for Anti-lock Braking Systems. In *Proceedings of the 15th International Conference on Cryptographic Hardware and Embedded Systems (CHES'13)*. Springer-Verlag, Berlin, Heidelberg, 55–72.

[39] Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013), 475–495.

[40] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Notices*, Vol. 40. ACM, 281–294.

[41] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. 2011. On the requirements for successful GPS spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 75–86.

[42] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. 2005. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems (TECS)* 4, 4 (2005), 779–818.

[43] Jiang Wan, Arquimedes Canedo, and Mohammad Abdullah Al Faruque. 2015. Security-aware functional modeling of cyber-physical systems. In *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE, 1–4.

[44] Armin Wasicek, Patricia Derler, and Edward A Lee. 2014. Aspect-oriented modeling of attacks in automotive cyber-physical systems. In *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 1–6.