[Unit 3](#_bookmark3)

## [Logical Correctness for Hybrid Systems](#_bookmark3)

1. A hybrid system is a dynamic system whose behavior changes both discretely and continuously.
2. Hybrid systems constitute a powerful formalism for understanding, modeling, and reasoning about a wide range of devices that affect our day-to-day lives.
3. These range from the simple, such as a room thermostat, to the critical, such as the airbag controller in a vehicle.
4. Hybrid systems also enable rigorous analysis to be performed that can verify and predict correct behavior of such devices.

#### [Introduction and Motivation](#_bookmark3)

* A key characteristic shared by many such devices is that they are built on digital computation tech
* Main feature of many such hybrid systems (e.g., pacemakers, vehicle airbag controllers, nuclear plant controllers) is that their incorrect operation can lead to catastrophic consequences.
* The combination of these two facts implies that verifying “functional correctness”—that is, the safe and secure operation—of hybrid systems has fundamental importance.
* Informally, a hybrid system can be viewed as a state machine having two types of transitions: 1)discrete jumps and 2) continuous evolution.
* A standard representation of a hybrid system is a hybrid automaton a type of finite-state machine in which **states represent discrete modes**, and transitions represent switching between the modes.
* Transitions are guarded by conditions on a set of continuous variables.
* The changes in the values of the continuous variables are specified via differential equations.
* The system stays within a mode only as long as a mode-specific invari ant remains satisfied by the continuous variables.
* Example:: for a thermostat that attempts to maintain the temperature between *m* and *M* by controlling a heater.
* It has two modes: *l*0, where a heater is turned off, and *l*1, where a heater is turned on. The system begins in mode *l*0 when the temperature *x* equals *M*.
* It remains in *l*0 as long as the invariant *x*  *m* for model *l*0 holds. At any point in time when the transition guard *x*  *m* holds, the system switches to mode *l*1.
* We focus on the class of functional correctness problems referred to as reachability. Informally, such a problem is specified by a formula describing a set of “good” states

*x*  *m*



*l*

0

*x*  *Kx x*  *m*

*l*1

*x*  *K*(*M*  *x*) *x*  *m*

*x*  *M*

*x*  *M*

**Figure 6.1:** *A hybrid automaton for a thermostat*

* The system is functionally correct if it never reaches a state that does not belong to .
* For example, a functional correctness property for the hybrid system in Figure is expressed by the formula *m*  *x*  *M*, which asserts that the temperature of the hybrid system always stays in the range [*m*, *M*].
* The Problem of Hybrid system is solved in 3 steps:
* **First, we categorize the reachability problem** for hybrid systems into **three** classes: discrete, real time, and fully hybrid.
* A discrete reachability problem is one that can be solved by ignoring completely the continuous dynamics of the system.
* A real-time reachability problem can be solved only by considering clocks and the passage of time.
* A fully hybrid problem requires reasoning about the dynamics of one or more continuous variables in addition to clocks.
* **Second, we restrict our attention to solutions** for hybrid system reachability that are based on an exhaustive, automated, and algorithmic veri fication technique known as model checking
* **Third**, the practitioners who are interested in learning more about the state of the practice in applying model checking to **verifying correctness of hybrid systems**.

[**Basic Techniques**](#_bookmark3)

Two types of hybrid systems verification are **discrete verification** and **temporal logic**, which has provided the foundation for the rest of the techniques that have provided multiple examples and the tools available to the practitioner.

##### [Discrete Verification](#_bookmark3)

* A discrete event system (DES) is a dynamic system that behaves in accordance with an abrupt occurrence, at possibly unknown irregular intervals of physical events
* It contains discrete states, whose state changes are event driven; that is, the state evolution depends entirely on the occurrence of asynchronous discrete events over time.
* Discrete event systems are found in a number of domains, including manufacturing, robotics, logistics, computer operating systems, and network communication systems.
* They require some form of control to ensure an orderly control of events for the purpose of eventually achieving some (good) behavior or always avoiding some (bad) behavior.
* Eventually achieving a good behavior is termed a **liveness property**, whereas always avoiding a bad behavior is termed a **safety property**.
* The capability to design and analyze discrete event systems is based on the fact that they can be modeled (represented) in an automata for- malism.
* The representations are abstractions of the actual system. Formalisms utilize precise semantics to express behavior, with typical examples including state charts, Petri nets, Markov chains.

###### Model Checking Tools and Associated Logic

* The models used in model checking are formal representations of system behavior.
* State charts are one formal representation that is widely used.
* Claims are formal specifications of expected properties of the system that one is attempting to verify.
* The formal definition of models and claims enables automated tools to verify whether particular claims hold against a model.
* If a claim is well written and the model is a faithful representation of the system, the verification using model checking will indicate whether the system possesses or does not possess the expected property represented by the claim.
* Depending on the kind of expected property and the type of model checker, different notations can be used to formalize the claim.
* **Temporal logic** is an extension to propositional logic that considers time; it is a formal approach for specifying and reasoning about the dynamic characteristics of a system.
* Temporal logic has a sequence of states. These sequences of states (traces) can be either finite *s*0 , *s*1 , *s*2 ,, *sn*  or infinite *s*0 , *s*1 , *s*2 , . States rep- resent finite time intervals of fixed conditions for a system.
* Two prominent forms of temporal logic are distinguished **Linear temporal logic** (LTL) and branching temporal logic, which includes **computational tree logic**  (CTL).
* The difference between the two lies in the conceptualization of unfolding time. In linear temporal logic, the evolution of time is viewed as linear—that is, a single infinite sequence of states. In contrast, CTL views the evolution of time as possibly proceeding along a multiplicity of paths (i.e., branches). Each path is a linear sequence of states.

###### Linear Temporal Logic

* Let *AP* be a set of “atomic propositions.” A LTL formula  satisfies the following BNF grammar:

 :T| *p* |  |    |    | *X*  |  *U*  | *G*  | *F* 

* Here T denotes “true”, *p*  *AP* is an atomic proposition,  denotes
* logical negation,  denotes conjunction,  denotes disjunction, *X* denotes the next-state temporal operator, *U* denotes the “until” tempo- ral operator, *G* denotes the “globally” temporal operators, and *F* denotes the “eventually” temporal operator.
* For example, let *AP*  {*p*, *q*, *r*}. Then some possible LTL formulas are as follows:

1  *G* *p*  *XFq*

2  *G* *p*  *XF* *q*  *r*

* As we will describe in the next section, every LTL formula denotes a property over paths (i.e., sequences of system states). Informally, 1 holds on a path if globally (i.e., at each state of the path) either the proposition *p* holds, or from the next state, *q* holds eventually. Similarly,
* 2 holds on a path if globally (i.e., at each state of the path) either the proposition *p* holds, or from the next state, *q* or *r* holds eventually.
* An LTL formula is interpreted over **Kripke structures.**
* **Kripke structure**- is a finite-state machine whose states are labeled with atomic propositions.
* Formally, a Kripke structure *M* is a 4-tuple (*S*, *I*, *R*, *L*) where

*S* is a finite set of states;

*I* *S* is the initial state;

*R*  *S*  *S* is the transition relation; and

*L* : *S*  2AP is a mapping from states to atomic propositions.

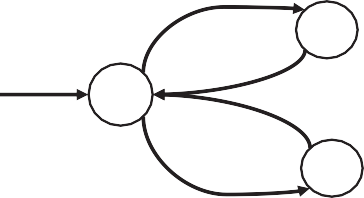
*L* maps each state to the set of atomic propositions that are true in that state.

Figure below shows a Kripke structure *M*  (*S*, *I*, *R*, *L*) with three states:

*S*  {*s*0 , *s*1 , *s*2 } , where *s*0 is the initial state. The transition relation is shown by arrows; that is, *s*, *s* *R* if there is an edge from *s* to *s* in the figure.

Each state *s* is labeled with *L* *s*2   {*q*}.

*L*(*s*); that is,



*r*

*s*0

*p*

*s*

1

*s*2

*q*

**Figure** *An example Kripke structure with AP*  *p*, *q*, *r**L* *s*0   {*p*},

*L*  *s*1  {*r*}, and

* A trace of a Kripke structure *M*  *S*, *I*, *R*, *L* is an infinite sequence of states *s*0 , *s*1 , *s*2 , such that

*s*0  *I*  *i*  0.*si* , *si*1  *L*

* A trace begins with the initial state and then follows the transition relation infinitely often.

*i*

* Given a trace  *s*0 , *s*1 , *s*2 ,, we write *i* to mean the infinite suffix of  starting with *s* . Then we say that
  + satisfies a LTL formula , denoted  |= , if the following holds:
*  T
*  AP, then  L(*s*0 ); that is, the first state of  is labeled with 
*    , then  $  
*   1  2 , then  |= 1 and  |= 2
*   1 *U* 2 , then there exists a *k*  0 such that

0  *i*  *k*. *i* |=    *k* |= 

1 2

Note that , F, and G can be expressed in terms of the other operators:

1  2  1  2 

*F*  T *U*

*G*  (*F* )

Finally, a Kripke structure *M* satisfies an LTL formula , denoted *M |=*  , if for every trace  of *M*,  *|=* .

Let M= kripke structure of the given figure,

*1*  *G* *p*  *XFq* and 2  *G**p*  *XF* *q*  *r* . Then, *M !|=* 1 since for

the trace   *s*0 , *s*1 , *s*0 , *s*1 ,, we have that  !*|=* 1 (that is, we cannot verify that proposition 1 evaluates to true for trace  ). On the other hand, *M* |= 2.

**Example: Helicopter Flight Control Verification**

A typical helicopter is composed of a number of computer-based subsystems, as shown in Figure .

Ethernet

RS-422

Air Data Computer (ADC)

Communi- cations Digital Radio (COM)

Aircraft Network Manage- ment (ANM)

Mission Processor (MP)

Multi- function Display (MD)

**Figure :** *Computer-based subsystems of a helicopter*

Navigation (NAV)

Flight Manager (FM)

Stabilizer

Collective Grip

Aircraft System Manage- ment (ASM)

* The basic subsystems of a typical helicopter include the aircraft sys- tems management (ASM), mission processor (MP), multifunction dis- play (MD), communications (COM), navigation (NAV), flight controls (FC), air data computer (ADC), and flight manager (FM).
* These subsystems communicate over redundant serial buses such as Ethernet, RS-422, or Mil-1550. From an operational perspective, the pilot controls the helicopter through controls in the cockpit (cyclic, collective flight grip).
* The controls are interfaced to the FM, and their states are com- municated to their subsystems via the serial buses. The pilot receives information about the helicopter’s status via the multifunction display. Using the hard and soft keys of the MD, the pilot can also initiate various flight modes, status displays, navigation displays, and so on.
* In many current helicopter designs, the majority of the controls are “fly by wire,” meaning that the control devices contain electronic sensors that convert positional or force quantities into electronic signals that are read by the subsystem computer, and acted upon via the embedded control logic.
* The subsystem computer also interfaces to the appropri- ate actuators, which adjust such things as motor speed, rotor blade pitch, stabilizer angle, and so on.

###### Verification of Operational Requirements

* The requirements are derived in part from subject-matter experts (SMEs). ----the pilots.
* The verification activity can follow the requirements decomposition. The decomposed requirements are then mapped onto the appropriate subsystems and, in many cases, across subsystems
* In our example, we focus on the operation of the helicopter’s stabilizer.
* The behavior of the helicopter, and the perspective taken is that of the pilot.

###### Functionality of the Helicopter Stabilizer

* Helicopters are inherently complex and unstable rotorcraft.
* The rotational moments generated by the main rotor can cause the body of the helicopter to rotate around the rotor axis.
* The tail rotor acts to counteract the horizontal rotational moments. During horizontal flight, a combination of the rotational moments in conjunction with thrust generated by the main rotor can make the helicopter unstable at certain airspeeds.
* This instability usually results in the rising of the helicopter’s tail, which under certain situations can cause the helicopter to “nose over.”
* A stabilizer airfoil (“stabilizer,” for short) changes the direction of the airflow in the tail area and forces the tail down at higher air speeds.
* The stabilizer interfaces with the flight manager that controls, through a linear motor actuator, the horizontal stabilizer flight surface. The flight manager has a control algorithm that operates in two modes: **automatic** and **manual**.
* The pilot interacts with the stabilizer in two ways: (1) by choosing the operational mode of the stabilizer position control system (auto or manual) or (2) in manual mode, by directly interacting with the stabilizer actuators.
* The interaction point is the collective flight grip, which incorporates a switch for control of the stabilizer.
* This three-position switch allows the stabilizer to be moved throughout its operational range and to be reset to the automatic mode.
* The nose up/nose down (NU/ND) switch on the flight grip enables the pilot to issue nose up/ down commands as well as reset commands to the stabilizer.
* In addition, the pilot may request entry into nap of earth (NOE) mode by manipulating the multifunction pilot display. NOE mode is a flight mode that allows the helicopter to fly close to the earth—for example, during search operations. Flying close to the earth in a helicopter can affect the stability of the aircraft under certain conditions; hence height above the ground is part of what describes the NOE mode.
* Preliminary helicopter design documents provide further details about the expected operational modes and the conditions under which transitions can occur between the modes.
* The design documents indi cate that the system always starts in automatic mode.
* The pilot can explicitly select manual mode or NOE mode, and can “reset to” automatic mode from both the NOE and manual modes.
* Some transitions are initiated by the pilot, whereas others are initiated by environmental sensors or are triggered by system failure modes.
* Using this information, we capture the operational modes and transitions, initially without taking failure modes into account, and express them in a state chart.
* This allows us to formulate claims that can be verified against the detailed logic design of the stabilizer, which is ultimately realized as code within the FM.
* The state chart Figure of the helicopter operations shows pilot-initiated transitions as solid lines, and airspeed- triggered transitions as dashed lines.

###### Claim Generation

* By studying the state diagram, we can begin to develop claims regarding system operation that must be proved.
* As an example, the initialization of the system places it in auto-low-speed mode.
* Another example would be that manual mode can be entered only from the auto-low-speed or NOE mode. Following is a set of the claims that can potentially be checked:

initial

Airspeed < = 80.0 KTAS

**Auto**



**Auto**

HiSpeed

LoSpeed Pilot NOE Cmd

Pilot Reset Command

Airspeed > 80.0 KTAS

**Manual**

**NOE**

Pilot NU/ND Cmd (ND/NU Enabled)

Airspeed represented by VC\_SEL in SRS

**Figure :** *State diagram showing stabilizer modal behavior*

* + The system always begins in auto-low-speed mode. Suppose there is a proposition *AutoLowSpeed* that labels every state where the sys- tem is in auto-low-speed mode. This claim is then captured by the LTL formula   *AutoLowSpeed*. According to the semantics of LTL,

 is satisfied if the initial state of the system satisfies *AutoLowSpeed*.

* + The system can never go into auto-high-speed mode until airspeed

> 50 knots. (In the rest of this chapter, we use knots and KTAS syn- onymously.) Suppose there are two propositions: *AutoHighSpeed*

and *AirSpeedGT* 50. This claim is then expressed in LTL as

*G* *AutoHighSpeed*  *AirSpeedGT* 50.

* + If the helicopter is in auto-high-speed mode and airspeed  50 knots, the system will go to auto-low-speed mode.
  + If the helicopter is in auto-low-speed mode and NOE command = true, then the system will go into NOE mode.
  + If the helicopter is in auto-high-speed mode and airspeed > 50 knots, then it will never be the case that the system will enter into NOE mode.
  + If the helicopter is in auto-high-speed mode, airspeed  50 knots, and the pilot selects NOE mode, the system will eventually go into NOE mode. Suppose there are two propositions: *NOECommanded* and *NOEMode*. This claim is then expressed in LTL as *G*(*AutoHighSpeed*  *AirSpeedGT* 50  *NOECommanded*  *F NOEMode*) .
  + When the helicopter is in NOE mode, the stabilizer position will be commanded to become –15 degrees trailing edge down.
  + When the helicopter is in stabilizer manual mode, the stabilizer will eventually reach its commanded position.

These claims represent behavior under normal operating conditions (the list is not intended to be exhaustive). The effects of various compo- nent failures could be overlaid on the modal model, and claims regard- ing operation given failures could be generated and verified.

* To perform verification, the design of the logic for the flight manager must be captured using Simulink Stateflow state charts;
* We are verifying that the logic involved in the control of the stabilizer supports the claims included in the verification set.
* The underlying logic is translated into software that runs on the target processor.
* If the translation of the logic into code is performed correctly, then the execution of the code will support the behavior of the higher-level state chart.
* Hence another round of verification will be necessary when the applications are run on the target processor.

###### Top-Level State Charts

* In our example, the stabilizer logic itself has been decomposed into 13 substates and each substate is invoked by a 13-step sequencer.
* Each substate contains logic in the form of flowcharts that express the logic flow of the stabilizer control.
* A generic depiction of the 13-substates chart designed with the Simulink state chart designer is shown in Figure below.
* We will now look at a smaller subset of the stabilizer control logic that includes the sequencer, Stabilizer Control Logic #1 (SCL1), and Stabilizer Control Logic #2 (SCL2). Figure below shows these two logic blocks.
* The state charts in Figure **A** indicate the **input signals** (variables), which are aligned on the right side of the ovals, and the **output signals** (variables), which are aligned on the left side of the ovals.
* Signal flow begins at the left and goes to the right.
* The signals are user-defined variables that are either in local scope to the stabilizer itself or global variables outside the scope of the stabilizer.
* The signal can be either a Boolean, integer, or floating-point value and is defined within Simulink.
* The signal 1 from the sequencer is asserted, the logic contained within SCL B1 begins its execution.
* The values in the input signals are read, the internal logic is executed, and the output signals are written to as the execution proceeds.
* The next tick of the sequencer, signal 2 is asserted (signal 1 is deasserted), the input signals are read, the internal logic of SCL B2 is executed, and the output variables are written as the logic is executed.

This execution sequence continues for each of the 11 remaining SCL

blocks, and then the cycle repeats.

Minor Cylcl Clock

Stabilizer Control 1

Stabilizer

Control 2

Stabilizer

Control 3

Stabilizer

Control 3B

Scheduler

Stabilizer

Control 3C

Stabilizer Control 4

Stabilizer Control 5

Stabilizer Control 6

Stabilizer Control 6B

Stabilizer Control 7

Stabilizer Control 8

Stabilizer Control 9

Stabilizer Control 10

CL1 CL2 CL3 CL3B CL3C CL4

CL5 CL6

CL6B CL7 CL8 CL9 CL10

**Figure :** *Simulink state charts of the stabilizer control logic*

* The importance of this observation is that the more extensive the use of global variables is, the more the system states contribute to the behavior of the stabilizer.
* The number of states may become so large that the memory in the computer system cannot hold all of the states necessary for the model-checking algorithms to work—a phenomenon known as the **state explosion problem.**
* To overcome this problem new techniques need to be followed.

minor cycle clock

1 COL\_A\_ NOK

call()

PATH

13

STAB\_LAG0\_INIT

SLM

call() STAB\_LAG0\_INIT

PATH



SLM

AS1 AS2 DEG\_A

1

AS1

3

2

AS2

GFADE >

CL1 CL2

CL3 CL3B CL3C CL4

CL5 CL6

CL6B CL7

CL8 CL9 CL10

COL\_A\_NOK

2

COL\_B\_NOK

3

SELSTAB\_AUTO

4

SB1PWR\_NOK

5

SB2PWR\_NOK

6

SBNOSDNBP

7

SBNOSUPBP

11

AUTO\_MODE\_FAIL

12

MAN\_MODE\_ON

8

A1\_AUTO\_FAIL

9

COL\_B\_NOK SELSTAB\_AUTO SB1PWR\_NOK SB2PWR\_NOK SBNOSDNBP SBNOSUPBP AUTO\_MODE\_FAIL MAN\_MODE\_ON A1\_AUTO\_FAIL

A2\_AUTO\_FAIL

D0MQ SC1 SC3 LAGK1 LAGK2 LAGK3 SSPD1 SSPD3 SSPD4

AIR\_SPD

SSPD4

SSPD3

SSPD1

LAGK3

LAGK2

LAGK1

SC3

SC1

DEG\_B AS0 AUTO\_MODE\_FAIL > MAN\_MODE\_ON > NOE\_ENGAGED

NOE\_ON PS1EXT\_CMD PS1RET\_CMD PS2EXT\_CMD PS2RET\_CMD

RT1

DEG\_A

4

DEG\_B

5

NOE\_ENGAGED

6

7 NOE\_ON PS1EXT\_CMD

8

9 PS1RET\_CMD

PS2EXT\_CMD 10

11 PS2RET\_CMD

Scheduler

STABSPD7

A2\_AUTO\_FAIL

STABSPD7

AIR\_SPD

AUTO\_MODE\_FAIL\_OUT>

StabilizerControl\_1

TX\_SEL LAGV1 LAGV2 LAGV3

RT2 R\_NOE SC\_AUTO

SLAGS\_INIT\_OUT

S1E\_CMD

RT1

12

13 RT2

R\_NOE

14

SC\_AUTO

15

SLAGS\_INIT\_OUT

16

10

STABSPD8

AIR\_SPD

16

TX\_SEL

S1R\_CMD S1AUTOS\_CMD

S2E\_CMD

S2R\_CMD S2AUTOS\_CMD

17

S1R\_CMD

19

S2E\_CMD

21

S1E\_CMD

18

S1AUTOS\_CMD

20

S2R\_CMD

1/Z

SCMB1 SRT1 SRT2 LAGV1

LAGV2\_OUT LAGV3\_OUT

Stabilizer Control\_2

S2AUTOS\_CMD

22

SRT1

1/Z

23

SRT2

1/Z

211

6.2 Basic Techniques

**Figure A:** *Stabilizer control logic blocks 1 and 2*

###### Detailed State Charts

* Figure **B** shows the input and output signals of SCL1, and Figure **C** shows the login in the form of a flowchart that is contained within SCL1.
* The input signals on the left of the top-level state chart are a com- bination of global variables (list) and local stabilizer variables (list)..
* The control logic is produced as part of the detailed design activity of the project.
* The mapping takes the comparison, assignment, and math blocks and turns them into states, while the operations within the blocks become the triggers on the arcs between the states.
* The transformation for SCL1 is shown in Figure **C**.
* The underlying Simulink simulation engine can take this Stateflow diagram and, by assigning values to the variables, can use the underlying model-checking engine to verify or falsify the claims.
* With this understanding of the modeling semantics of Simulink Stateflow chart, we can now look at how claims can be constructed using System Design Verifier and analyzed.
* **Functional requirements** for discrete systems are typically explicit statements about expected **liveness properties** (e.g., behaviors that a system exhibits) and **safety properties** (e.g., behaviors that it must never exhibit).

###### Formalizing Claims in System Design Verifier

* **Simulink Design Verifier** (**SDV**) can express formal requirements using MatLab functions and Stateflow blocks.
* Each requirement in Simulink has one or more verification objectives associated with it.
* These verification objectives are used to check whether the design matches the specified properties (claims).
* The block library provided in SDV includes blocks and functions for test objectives, proof objectives, assertions, constraints, and a dedicated set of temporal operators for modeling of verification objectives with temporal aspects.
* In SDV, a claim is termed a “property” and is equivalent to a requirement.
* Proof of a property in SDV can be expressed in two General forms (1) a signal (variable) in the Simulink model that attains a

particular value or range of values during a simulation and (2) a signal (variable) in the Simulink model that is proved to be logically equivalent to an expression over a number of input and output signals.

1

COL\_A\_NOK

2

COL\_B\_NOK 3 SELSB\_AUTO 4

SB1PWR\_NOK 5 SB2PWR\_NOK 6 SBNOSDNBP 7 SBNOSUPBP

11

COL\_A\_NOK call() COL\_B\_NOK

SELSB\_AUTO

SB1PWR\_NOK PATH

SB2PWR\_NOK SBNOSDNBP SBNOSUPBP

AUTO\_MODE\_FAIL

8

A1\_AUTO\_FAIL 9 A2\_AUTO\_FAIL

AUTO\_MODE\_FAIL 12 MAN\_MODE\_ON

MAN\_MODE\_ON A1\_AUTO\_FAIL A2\_AUTO\_FAIL STABSPD7

AUTO\_MODE\_FAIL\_OUT

10

213

6.2 Basic Techniques

AIR\_SPD

**Figure 6.7:** *Input and output signals of SCL B1*

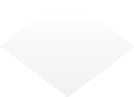
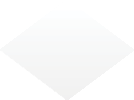
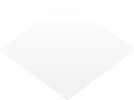
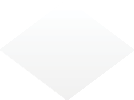
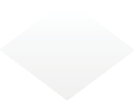
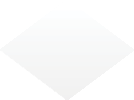
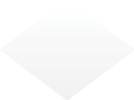
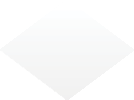
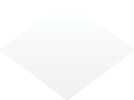
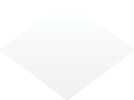
AIR\_SPD

Stabilizer Control Block #1

214

Chapter 6

Logical Correctness for Hybrid Systems



Start

NOT ((A1\_AUTO\_FAIL) OR (A2\_AUTO\_FAIL))

T

T

T

T

NOT(COL\_A\_NOK OR COL\_B\_NOK)

NOT (SB1PWR\_NOK)^NOT (SB2PWR\_NOK)

SELSB\_AUTO

F

F

F

F

T

NOT (AUTO\_MODE\_FAIL)

T

AIR\_SPD>= STABSPD7

T

MAN\_MODE\_ON

F

F

F

T

T

T

NOT

(MAN\_MODE\_ON) NOT(SBNOSDNBP) NOT(SBNOSUPBP)

F

F

F

SP = 2

EXIT

SP = 1

SP = 4

SP = 2

AUTO\_MODE\_FAIL=T

SP = 3

**Figure 6.8:** *A flowchart of the logic contained with SCL B1*

{AUTO\_MODE\_FAIL\_OUT = AUTO\_MODE\_FAIL}



{!(A1\_AUTO\_FAIL || A2\_AUTO\_FAIL)} {!(COL\_A\_NOK||COL\_B\_NOK)} {!(SB1PWR\_NOK) &&

!(SB2PWR\_NOK)}

{SB\_AUTO}

1 1 1 1

2 2 2

2

{!(AUTO\_MODE\_FAIL)} {AIR\_SPD >= STATSPD7} 1 1

{MAN\_MODE\_ON} 1

2

{AUTO\_MODE\_FAIL = true}

2

2

{!(MAN\_MODE\_ON)} {!(SBNODNSP)} {!(NOSUPSP)}

{SP = 3}

1

2

{SP = 1}

2 1 2 1

{SP = 2}

{SP = 4}

215

6.2 Basic Techniques

**Figure 6.9:** *The Simulink Stateflow chart for SCL1*

* The model is simulated by providing all combinations of inputs and logging the corresponding outputs;
* SDV then checks whether the proof that is being checked is valid (true) for all possible combinations of inputs.
* SDV provides a report ind cating that the proof to be verified either is satisfied (true for all combinations) or failed (false), in which case a counter example is provided. The counter example is the combination of inputs that make the proof false.
* In Simulink, a property is a requirement that one models in Simulink, Stateflow, and MatLab function blocks.
* A property can be a simple requirement, such as a signal in the model that must attain a particular value or a range of values during simulation.
* The SDV software provides two blocks that allow one to specify property proofs in Simulink models. *Proof objective* blocks define the values of a signal to prove, and *proof assumption* blocks constrain the values of a signal during a proof.
* We will use the proof objective block to prove the NOE mode claim. The claim we will review basically states that if the system is in auto- high-speed mode and the air speed is greater than 50 knots, the system should never go into NOE mode.
* Thus, we must insert a proof objective block on the NOE ON signal in the Simulink model.
* In addition, because the auto-high-speed mode comprises a number of signals, we must use some logic blocks to construct the claim.
* To construct a logical expression of signals that constitute the claim, we use a combination of MatLab logic blocks, which lead to the expres- sion shown in Figure **D**.
* The auto-high-speed mode is a collection of signals that include A1\_AUTO\_FAIL, A2\_AUTO\_FAIL, COL\_A\_NOK, COL\_B\_NOK, SB1PWR\_NOK, SB2PWR\_NOK, SELSB\_AUTO, AUTO\_MODE\_FAIL, and MANUAL\_MODE\_ON.
* These signals are inputs to the MatLab function block named Auto Mode HS.
* The airspeed variable is also an input, but it needs to be compared to a constant and converted to a binary value that indicates whether it is above or below a threshold (in this case, 50 knots).

9 A1\_Auto\_FAIL

A1\_Auto\_FAIL

A2\_Auto\_FAIL

COL\_A\_NOK

COL\_B\_NOK

SB1PWR\_NOK

SB2PWR\_NOK

SELSB\_AUTO

AIR\_SPD

8 A2\_Auto\_FAIL

1 COL\_A\_NOK

2



COL\_B\_NOK

4



NOT

AND

SB1PWR\_NOK

AIR\_SPD

5

SB2PWR\_NOK



3 SELSB\_AUTO



Compare

> 80

to constant

Auto mode HS

11

AUTO\_MODE\_FAIL1 Logical

NOT

operator1

12

MANUAL\_MODE\_ON1 Logical

operator2

Logical operator A B

true

A-->B P

call()

AS1 A

AS2 B

DEG1 D

DEG2 D2

AS3

NOT

Logical operator3

6

NOE\_ON

Implies

Scope

AUTO\_MODE\_FAIL> MANUAL\_MODE\_ON>

NOE\_ENGAGE NOE\_ON

5

ENGAGE

**Figure D:** *Construction of the proof objective for NOE mode*

The following logic is contained within the Auto Mode HS function block:

function y = fcn(A1\_AUTO\_FAIL, A2\_AUTO\_FAIL, COL\_A\_NOK, COL\_B\_NOK, SB1PWR\_NOK, SB2PWR\_NOK, SELSB\_AUTO, AIR\_SPD)

%#codegen

y = ~(A1\_AUTO\_FAIL || A2\_AUTO\_FAIL)…

&& ~(COL\_A\_NOK || COL\_B\_NOK)…

&& (~(SB1PWR\_NOK)&& ~(SB2PWR\_NOK)…

&& AIR\_SPD)… && SELSB\_AUTO;

* This logic expression equates to the conditions that make up the auto mode and the airspeed.
* The output of this block is connected to a three- input AND logic block that ANDs in the AUTO\_MODE\_FAIL1 signal and the MANUAL\_MODE\_ON1 signal.
* The conjunction of these signals feeds an IMPLIES logic block, which tests whether the A input implies the B input. The block outputs a Boolean value of false when the A input is true and the B input is false; otherwise, it outputs true. The truth table is shown in Figure E.

###### Running the Model-Checking Engine

The model is compiled and checked for compatibility with SDV, and then the property prover is run.

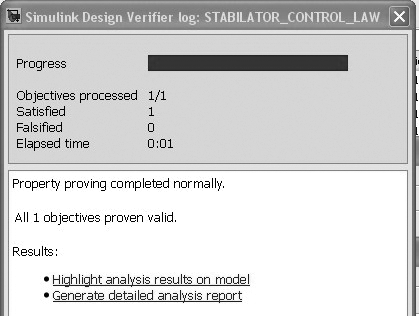
1. Two outcomes are possible: the objectives are proven valid (e.g., the claim is proven true), or
2. The objectives are shown to be invalid. The outcome of this exercise in our example is that the objective is proven valid. The output window is shown in Figure F.

The entire stabilizer model consisted of 13 state charts, with each state chart having an average of 15 variables.

Over the entire stabilizer model, approximately 25 of the variables are global system variables.

|  |  |  |
| --- | --- | --- |
| A | B | Output |
| F | T | T |
| T | F | F |
| F | F | T |
| T | T | T |

**Figure E:** *IMPLIES block truth table*



**Figure F:** *SDV output showing the NOE mode claim to be true*

###### Observations

* + - The combination of Simulink and Design Verifier provides a fairly powerful approach to incorporating a formal model-checking verifica tion activity into the system design.
    - The tool set can handle large systems at abstract levels and can check detailed logic designs as shown in the previous example.
    - As in the case with all model-checking tools, there is a limitation that is encountered known as the **state explosion problem.**
    - Consider that there may be *m* systems and each system can have *n* states; then the asynchronous composition of these systems will have *nm* states.
    - Suggested approaches to mitigate the problem within SDV are to limit the scope of variables used in the design of various subsystems and to use the technique called assume-guarantee reasoning.
    - Other limitations to the tool reflect the fact that it is limited to safety properties only.
* The tool cannot use LTL or CTL expressions directly in the proof stage. While this capability may be developed as the tool evolves, the best that can be done now is a loose mapping of LTL expressions into the semantics of the proof prover.
* Although Simulink has the capability to generate C code, the user must rely on the accuracy of the translation to ensure correctness.

###### For the Practitioner

* + The tools within Simulink provide some advantages to the practitioner wishing to use formal methods in a design.
  + Working within the MatLab–Simulink development environment allows many levels of design detail to evolve while maintaining traceability to the top-level design description.
  + The capability of simulating and testing a design complements the formal model checking.
  + In addition, through the use of hardware in the loop testing approaches, effects of timing can be observed on the design.
  + In dealing with systems of any size, the practitioner would benefit from developing or using automated means to help identify and man- age claims, perhaps through the use of a stylized grammar for express- ing functional and nonfunctional requirements.
  + It may then be possible to develop a set of common proof block templates using the primitive blocks contained within Simulink and SDV.
  + In addition, developing a technique to manage variable scoping during design would help reduce the potential for state explosion.

###### For the Researcher

* The state explosion problem has been the focus of a great deal of research over the last 10–15 years.
* The nature of this problem is such that as larger and larger systems are developed and need to be verified, the number of states will continue to grow.
* Given that the flowchart representation is an abstraction above the generated code, developing techniques to verify the code and relate it back to the higher-level abstraction is an area where research remains in its infancy.
* Additionally, developing approaches that consider the application code in conjunction with the operating system and guaranteeing its functional and temporal behavior seem to be areas where very little research has been done.

#### [Advanced Techniques](#_bookmark3)

##### [Real-Time Verification](#_bookmark3)

* This means verifying a hybrid system whose correctness depends not only on discrete state information, but also on the flow of time.
* In contrast to a discrete state, which changes instantaneously from one value to another, time flows continuously at a uniform rate.
* A timed automaton is a finite-state machine augmented with a set of “clock” variables.
* Within each state, each clock variable records the passage of time.
* Transitions between states are instantaneous, and guarded by constraints on the clock variables.
* A subset of the clock variables is reset during each transition.
* The choice of which syntax to use is guided by some other factor, such as suitability to the application domain, familiarity, and tool support.
* In this context, we use the syntax of the model checker UPPAAL, which is one of the most popular and actively maintained tools for verifying timed automata.

###### Example: Simple Light Control

Consider an intelligent light switch1 that has the following specifications:

* + - * 1. The switch has three states: off, medium, and high.
        2. Initially, the switch is in the off state.
        3. If the switch is pressed from the off state, it moves to the medium state.
        4. If the switch is pressed from the medium state, there are two possible outcomes:

If the latest press occurred within 10 seconds of the previous press, then the switch moves to the high state.

If the latest press occurred more than 10 seconds after the previous press, the switch moves to the off state.

* + - * 1. If the switch is pressed from the off state, it moves to the on state.
* Clearly, time plays a crucial role in determining the behavior of this switch, if the switch is in the medium state, given the same external input (press), the next state of the switch is determined by the amount of time that has passed since the last input.
* In addition, any model of the switch must have some mechanism to keep track of the **amount of time elapsed** and express **conditions on time**.
* As expected, there are three states of the automaton—denoted as OFF, MEDIUM, and HIGH—with OFF being the initial state.
* In addition, an input action press? denotes the switch being pressed, and there is a clock variable clk.
* Each transition has a label of the form **guard[action]**, where  is the action triggering the transition, guard is a constraint on clock variables that must be TRUE for the transition to occur, and action is a set of assignments resetting certain clock variables.
* Note that {} means that guard is TRUE, and [] means that no clock variable is reset.
* In the Figure all transitions have labels such that  is press?, indicating that transitions happen only if the switch is pressed.
* The transition from OFF to MEDIUM happens at any time but resets variable clk. This captures specification 3.
* The transition from MEDIUM to HIGH requires that clk be no greater than 10, which captures specification 4a.

*press?* {*clk* > 10} [ ]

*OFF*

*MEDIUM*

*HIGH*

*press?* {}[*clk* := 0] *press?* {*clk* ≤ 10} [ ]

*press?* {}[ ]

**Figure :** *Timed automata to model an intelligent light switch*

* The transition from MEDIUM to OFF requires that the value of clk be greater than 10, which captures specification 4b.
* Both outgoing transitions from MEDIUM leave clk unchanged. Finally, the transition from HIGH to OFF can occur at any time, and leaves clk unchanged.

###### Composition and Synchronization

* Timed automata can be composed, which in turn means that a more complex timed automaton can be described in terms of simpler ones.
* A number of composition semantics have been studied in the literature.
* In this semantics, in addition to the natural synchronization provided by clocks, timed automata are synchronized via matching input-output action pairs that label their transitions. Consider again our example light switch from Figure 6.13.
* Recall that its transitions are labeled by the input action press?. Figure 6.14(a) shows the timed automata that models a “fast” user pressing the key every 3–5 seconds.
* Note that its transition is labeled by the output action press! and that it has its own clock variable uclk.
* The transition occurs only when 3  uclk  5 and resets uclk. The single state STATE of the automaton is also labeled with the “state invariant” uclk 5.
* This means that the automaton can remain in state STATE only as
* long as its clock is no greater than 5, in effect, modeling a user who waits no more than 5 seconds between successive switch presses.
* State invariants are useful to enforce “liveness” conditions; for example, the switch is pressed infinitely often, which is necessary to eliminate unre- alistic behaviors from the model.
* Figure 6.14(b) shows the timed automata that models a “slow” user who presses the key every 12–15 seconds.

*press*! {3  *uclk*  5} [*uclk* : = 0] *press*! {12  *uclk*  15} [*uclk* : = 0]

*STATE*

*uclk*  5

*STATE*

*uclk*  15

(*a*) (*b*)

**Figure 6.14:** *A timed automata modeling: (a) a fast user and (b) a slow user*

Note that its transition is also labeled by the output action press! and that it has its own clock variable uclk.

However, the transition occurs only when 12  uclk  15 and resets uclk.

The other difference from Figure 6.14(a) is that the state invariant for STATE is uclk  15.

###### Functional Properties

* In general, for timed automata, we can specify and verify functional properties that involve sequences of discrete state information and clock valuation.
* In the below Table shows a sampling of properties in UPPAAL syntax (column 2), and their informal meaning (column 3), for a system com- posed of the timed automaton shown in Figure.
* In the properties, Sw refers to the timed automaton from Figure.
* The last two columns in Table 6.1 show whether the property holds depending on which user automaton is selected from Figure.
* Column 4 shows the result for the “fast” system (using the user model from Figure (a).
* Column 5 shows the result for the “slow” system (using the user from Table).

1. Property 1 is FALSE for both systems because in each case, a press! action by the user forces the switch to transition away from the OFF state. Therefore, it is impossible for the switch to remain indefinitely in the OFF state for any possible system execution.

**Table:** *A Set of Functional Properties for the Timed Automata*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **UPAAL** | **Informal Meaning** | **Holds (fast)** | **Holds (slow)** |
| 1 | *E* [ ] *Sw*.*OFF* | Is it possible that the light switch is always OFF? | No | No |
| 2 | *E*  (*Sw*.*MEDIUM and Sw*.*clk*  6) | Is it possible that after the switch reaches the MEDIUM state, it is not pressed for more than 6 seconds? | No | Yes |
| 3 | *A*   *not deadlock* | The system does not deadlock? | Yes | Yes |
| 4 | *A*  *Sw*.*HIGH* | From any given state, the switch will always eventu- ally reach the HIGH state under all possible user behaviors. | Yes | No |
| 5 | *A*  *Sw*.*OFF* | From any given state, the switch will always eventu- ally reach the OFF state under all possible user behaviors. | Yes | Yes |

1. Property 2 is TRUE for the slow system because the slow user presses the switch only after every 12 seconds or more. However, the fast user presses the switch at least once every 5 seconds, forcing property 2 to be FALSE for the fast system.
2. Property 3 is TRUE for both systems. Note that the state invariants in the user models are critical for this result. Without these invariants, the timed automaton for the user can remain in state STATE indefinitely without generating any press! actions, causing the system to deadlock.
3. Property 4 is TRUE for the fast system because the rapid switch presses by the fast user force the switch to always visit the HIGH state after the MEDIUM state. In contrast, the property is FALSE for the slow system. Indeed, for the slow system, the HIGH state is never visited by the switch.
4. Property 5 is TRUE for both systems because the OFF state is visited infinitely often. Once again, the state invariants are crucial to ensure this property.

##### [Hybrid Verification](#_bookmark3)

* A fully hybrid system is a system with discrete and continuous parts such that continuous parts of the system evolve according to some differential equations.
* A real-time system, described in the previous section, is an instance of a hybrid system in which the only continuous variables are clocks that evolve at a fixed uniform rate.
* Formally, hybrid systems are represented by hybrid automata.
* A hybrid automaton is a finite-state machine with a set of continuous variables.
* Within each state, continuous variables record continuous changes relative to some time.
* The dynamics of the flow are represented by differential equations. Transitions between states are guarded by constraints on continuous variables.
* The ability to specify this kind of continuous change using differential equations makes hybrid automata much more powerful than the timed automata used for real-time systems.
* In particular, the reachability problem for hybrid automata—that is, whether a given state and a valuation of continuous variables are reachable—is undecidable. Thus, existing verification techniques are best-effort measures and require the user to specify the upper bound on the exploration depth.
* A number of notations have been developed for depicting hybrid automata and the tools to analyze them.
* The SpaceEx State Space Explorer tool from Verimag [(http://spaceex.imag.](http://spaceex.imag.fr/) [fr) is used for this verification.](http://spaceex.imag.fr/)

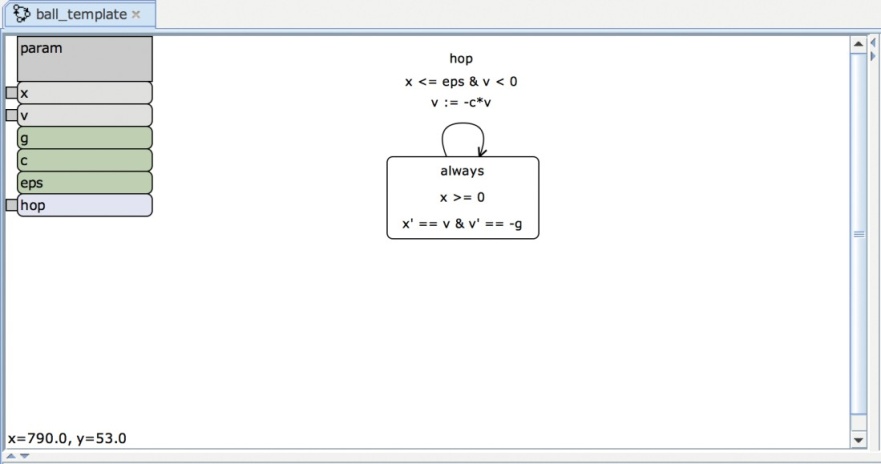
###### Example: Bouncing Ball

* We want to model a ball that is dropped from a fixed height. The model in SpaceEx is shown in Figure.
* It consists of two continuous variables: *x*, which represents the current height of the ball relative to the ground, and *v*, which represents the velocity of the ball.
* The system has a single discrete state in which the continuous flow is given by the fol lowing differential equations:

*dx*  *v* and

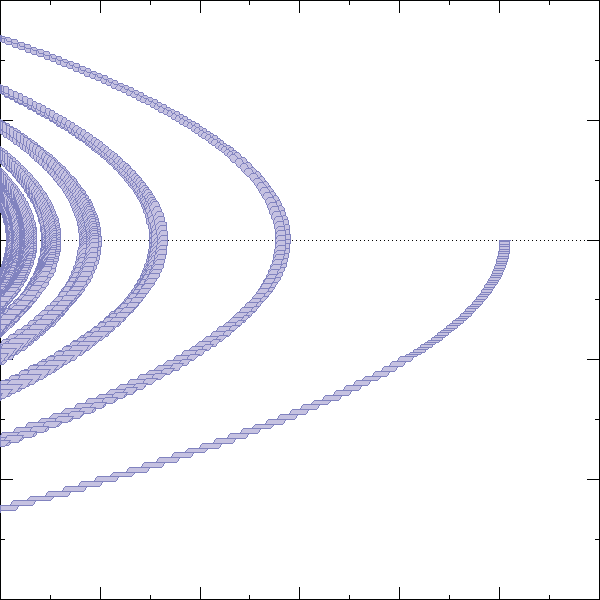
*dt*

*dv*   *g dt*



**Figure :** *A hybrid automaton for a bouncing ball example*

* Here *g* stands for standard gravity. *I*t is required that the distance to the ground is always positive.
* The system has a single transition that changes the continuous dynamics.
* Whenever the center of the ball is close to the ground (i.e., less than the parameter eps) and the velocity is negative, the ball hits the ground and shoots up with proportional velocity, dampened by factor c.
* Figure diagram shows the result of computing the set of reachable states using SpaceEx, where initially 10  x  10.2 and v  0.
* **Because the reachability of hybrid automata is undecidable in general, the user must specify how many reachability steps to apply for both discrete and continuous evolution of the system**.
* This example was constructed with 50 discrete steps and 80 continuous ones. The graph shows the height of the ball (*x*-axis) versus its velocity (*y*-axis).
* Initially the ball starts at a height of approximately 10 m and falls. The ground is hit at the velocity of slightly more than –4 m/s.
* Then, the ball bounces with the velocity of slightly less than 4 m/s because of the dampening effect of the impact. Each arc in the graph represents one complete bounce.

4

2

0



2

4

6 0

2 4 6

x

8 10 12

**Figure :** *Reachable states of the bouncing ball example*

###### Example: Thermostat

* This thermostat has two states: on and off.
* In the off state, the temperature gradually decreases.
* In the on state, it gradually increases.
* Whenever the temperature reaches a particular limit (too cold or too hot), the thermostat switches to the corresponding model.
* The model in SpaceEx is shown in Figure. The continuous dynamics in state off are given by the following differential equation:

*d temp*  0.1 temp

\*

*dt*

The continuous dynamics in the on state are given by the following:

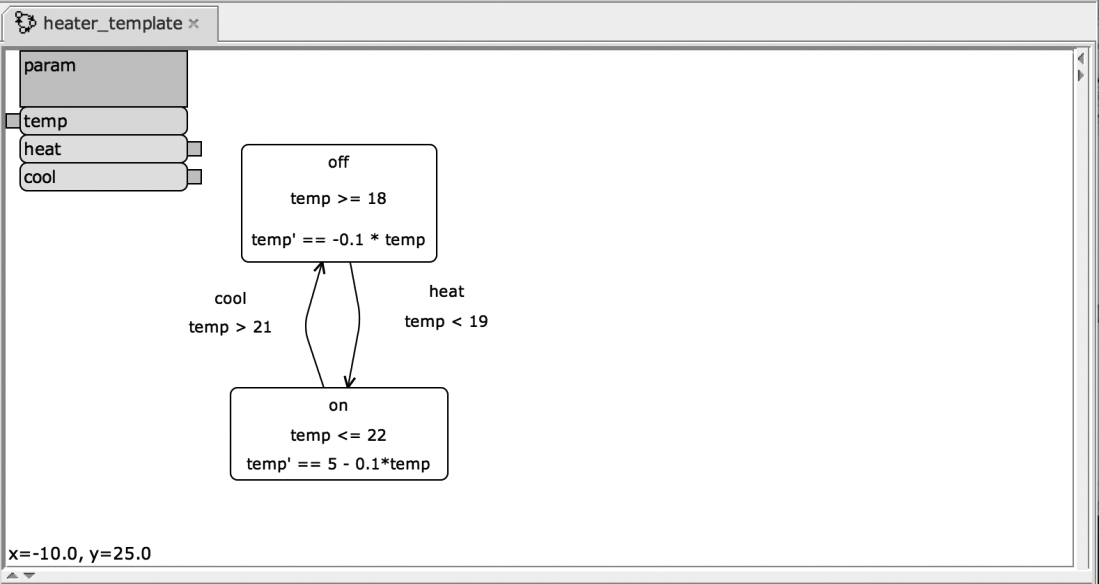
*d temp*  5  0.1 temp dt

\*

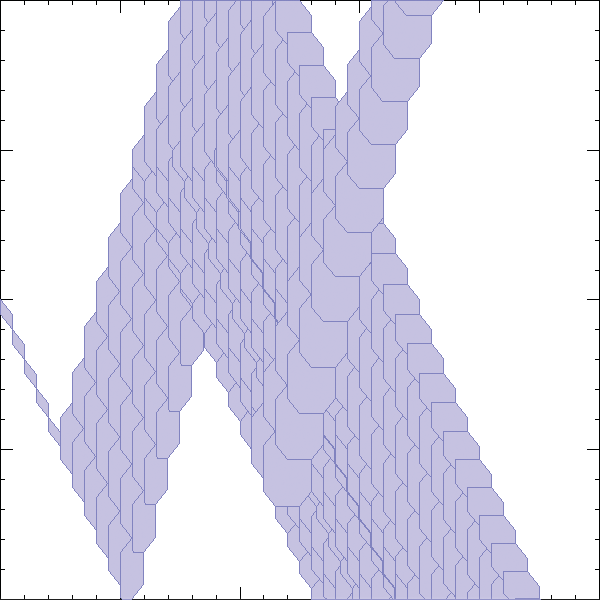
The set of reachable states computed by SpaceEx for the thermostat model is shown in Figure below.

The initial state was temp  20 and the system started in the off state.

The graph shows the possible values for the temperature (*y*-axis) relative to time (*x*-axis). As before, because the problem of computing all reachable states is undecidable, a limit for computation was selected: 40 discrete steps and 80 continuous ones. Note that this was sufficient to explore all the states the system can reach in approximately 5 seconds.



**Figure :** *A model of a thermostat*

22

21

20

temp

19

180

1 2 3 4 5

t

**Figure** *Reachable states of the thermostat example*

## [Security of Cyber- Physical Systems](#_bookmark3)

A wide variety of motivations exist for launching an attack on cyber- physical systems (CPSs), ranging from economic reasons (e.g., reduc- ing electricity bills), to pranks, to terrorism (e.g., threatening people by controlling electricity and other life-critical resources). The first-ever CPS malware called Stuxnet was found in July 2010. This malware, which targeted vulnerable CPSs, raises new questions about CPS secu- rity. CPSs are currently isolated, preventing external access to them. Malware, however, can spread using USB drives and can be specifically crafted to sabotage CPSs. The emerging, highly intercon- nected, and complex CPSs, such as vehicular networks, embedded medical devices, and smart grids.

237

#### [Introduction and Motivation](#_bookmark3)

In CPSs, cyber attacks can cause disruptions that transcend the cyber realm and affect the physical world.

Stuxnet is a clear example of a cyber attack that induced physical consequences. Conversely, pure physical attacks can also affect the security of the cyber system.

For example, the integrity of a smart meter can be compromised if the attacker uses a shunt to bypass the meter.

Likewise, a traffic signal preemption transmitter can compromise the integrity of a traffic light system. Placing a compromised sensor beside a legitimate one breaks the system secrecy while leaving the cyber system intact.

Based on the discussions at the Army Research Office workshop on CPS security in 2009, we classify current attacks on cyber-physical systems into four categories.

*Taxonomy of Attacks and Consequences in Cyber and Physical Systems*

|  |  |  |
| --- | --- | --- |
| **Consequence/Attack** | **Cyber** | **Physical** |
| **Cyber** | Eavesdropping on private information | Stuxnet CPS malware |
| **Physical** | Sensor bypassing | Instability due to physical destructions |

Defense mechanisms need to take into the account the dual nature of the CPS.

System theory-based approaches, which leverage the physical model of the system, need to be used in combination with cyber security to provide solutions for detection, response, reconfiguration, and restoration of system functionalities while keeping the system operating.

#### [Basic Techniques](#_bookmark3)

We analyze the cyber security requirements for cyber-physical systems, consider how the new attack models arise in the new CPS reality, and identify the basic corresponding countermeasures.

##### [Cyber Security Requirements](#_bookmark3)

In general, cyber security requirements for a system include three main security properties:

* *Confidentiality:* Prevents an unauthorized user from obtaining secret or private information
* *Integrity:* Prevents an unauthorized user from modifying the information
* *Availability:* Ensures that the resource can be used when requested
* **Confidentiality** of sensor data, such as meter data, global posi- tioning system (GPS), or accelerometer data, is clearly important.
* Confidentiality of control commands may not be important in cases where they are public knowledge.
* Confidentiality of software should not be critical, because the security of the system should not rely on the secrecy of the software, but only on the secrecy of the keys, according to Kerckhoffs’s principle.
* The integrity of sensor data and commands is important as well as the integrity of software.
* Compromised software or malware may potentially be able to control any device or component in the CPS.
* Denial-of-service (DoS) attacks are resource consumption attacks that send fake requests to a server or a network.
* Distributed DoS (DDoS) attacks are accomplished by utilizing distributed attacking sources such as compromised smart meters, traffic lights, or sensors in a vehicle.
* In CPS, availability of information is the most important aspect of the system’s operation.
* Availability of control commands is also important—for example, when reducing the speed of a vehicle to maintain a safe distance from the car ahead.
* By comparison, availability of sensor data (e.g., gas mileage) may not be as critical because the data can usually be read at a later point.
* Table below summarizes the relative importance of data, commands, and software. In this table, high risk implies that a property of certain information is critical, medium risk implies some importance, and low risk is noncritical. This classification enables prioritization of risks, to focus effort on the most critical aspects first.
* For example, the integrity of control commands is more important than the commands’ confidentiality; consequently, we need to focus on efficient cryptographic authentication mechanisms before addressing encryption.

##### [Attack Model](#_bookmark3)

To launch an attack, an adversary must first exploit entry points, and upon successful entry, deliver specific cyber attacks on the CPS.

**Attack Entry Points**

Unfortunately, the size and complexity of the networks in CPS provide numerous potential entry points:

* ***Inadvertent infiltration through infected devices****:* Malicious media or devices may be inadvertently infiltrated inside the trusted perimeter by personnel. For example, USB memory sticks have become a

*Importance of Security Properties for Commands, Data, and Software*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Control Commands** | **Sensor Data** | **Software** |
| **Confidentiality** | Low | Medium | Low |
| **Integrity** | High | High | High |
| **Availability** | High | Low | N/A |

popular tool to circumvent perimeter defenses.

* + *Network-based intrusion:* Exploiting poorly configured firewalls through both misconfigured inbound and faulty outbound rules is a common entry point, enabling an adversary to insert a malicious payload onto the control system.
  + *Backdoors and holes in the network perimeter:* Backdoors and holes may be caused by components of the IT infrastructure that have vulnera- bilities or misconfigurations. All of these kinds of network-based intrusions are particularly dangerous because they enable a remote adversary to enter the trusted control-system network.
  + *Compromised supply chain:* An attacker can pre-install malicious codes or backdoors into a device prior to shipment to a target location—a strategy called a supply chain attack.
  + Consequently, the need for security assurance in the development and manufacturing process for sourced software and equipment is critical for safe-guarding a cyber supply chain that involves technology vendors and developers.
* *Malicious insider:* An employee or legitimate user who is authorized to access system resources can perform actions that are difficult to detect and prevent. Privileged insiders also have intimate knowl- edge of the deployed defense mechanisms, which they can often easily circumvent.

###### Adversary Actions

Once an attacker gains access to the network, that person can perform a wide range of attacks.

###### Cyber Consequences

From the cyber point of view, a variety of consequences rooted in how software works may arise:

* *Spreading malware and controlling devices:* An adversary can develop malware and spread it so that it infects smart meters or company servers. Malware can be used to replace or add any func tion to a device or a system, such as sending sensitive information or controlling devices.
* *Vulnerabilities in common protocols:* CPS use existing protocols, which means they inherit the vulnerabilities of these protocols. Commonly used protocols include TCP/IP and Remote Procedure Call (RPC).
* *Access through database links:* Control systems record their activities in a database on the control system network, and then in mirror

*Types of Threats Created by Attacking Security Properties*

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Control Commands** | **Sensor Data** | **Software** |
| **Confidentiality** | Exposure of control structure | Unauthorized access to sensor data | Theft of propri- etary software |
| **Integrity** | Changes of control commands | Incorrect system data | Malicious software |
| **Availability** | Inability to control system | Unavailability of sensor information | N/A |

logs in the business network. A skilled attacker can gain access to the database on the business network, with the business network then providing a path to the control system network. Modern data- base architectures may allow this type of attack if they are config- ured improperly.

* + *Compromising communication equipment:* An attacker can potentially reconfigure or compromise some of the communication equipment, such as multiplexers.
  + *Injecting false information on sensor data:* An adversary can send packets to inject false information into the system.
  + *Eavesdropping attacks:* An adversary can obtain sensitive informa- tion by monitoring network traffic, which may result in privacy breaches or disclosure of the controlling structure of cyber-physical systems.

A SCADA protocol of noteworthy concern is the Modbus protocol, which is widely used in industrial control applications such as in water, oil, and gas infrastructures.

The Modbus protocol defines the message structure and communication rules used by process control systems to exchange SCADA information for operating and controlling industrial processes.

Modbus is a simple client–server protocol that was originally designed for low-speed serial communication in process control networks.

Given that this protocol was not designed for highly security-critical environments, several kinds of attacks are possible:

* + *Broadcast message spoofing:* This attack involves sending fake broad- cast messages to slave devices.
  + *Baseline response replay:* This attack involves recording genuine traf- fic between a master and a field device, and replaying some of the recorded messages back to the master.
  + *Direct slave control:* This attack involves locking out a master and controlling one or more field devices.
* *Modbus network scanning:* This attack involves sending benign messages to all possible addresses on a Modbus network to obtain information about field devices.
* *Passive reconnaissance:* This attack involves passively reading Modbus messages or network traffic.
* *Response delay:* This attack involves delaying response messages so that the master receives out-of-date information from slave devices.
* *Rogue interloper:* This attack involves attacking a computer with the appropriate (serial or Ethernet) adapters to an unprotected com- munication link.

###### Physical Consequences

The attacks mentioned here are not exhaustive, but they serve to illus- trate risks and can help developers ensure that grid systems are secure:

* *Interception of SCADA frames:* An attacker can use a protocol analy- sis tool for sniffing network traffic to intercept SCADA frames and collect unencrypted plaintext frames that may contain valuable information, such as source and destination addresses.
* *Malware targeting of industrial control systems:* An attacker can suc- cessfully inject worms into vulnerable control systems and repro- gram industrial control systems.
* *DoS/DDoS attacks on networks and servers:* An adversary can launch a DoS/DDoS attack against various components in the CPS, including networking devices, communication links, and servers.
* *Sending fake commands to systems in a region:* An adversary can send fake commands to a device or a group of devices in a target region.

##### [Countermeasures](#_bookmark3)

A number of countermeasures have been developed for these types of attacks.

###### Key Management

* Key management is a fundamental approach for information security. Shared secret keys or authentic public keys can be used to achieve secrecy and authenticity for communication.
* Authenticity is especially important to verify the origin of messages, which in turn is key for access control.
* The key setup in a system defines the root of trust.
* In a symmetric-key system, each entity and the trust center would set up shared secret keys and establish additional trust relationships among other nodes by leveraging the trust center, as in the Kerberos network authentication protocol[.](https://web.mit.edu/kerberos/)
* The challenge in this space is key management across a very broad and diverse infrastructure.
* Business, policy, and legal aspects need to be considered in setting up the key management system, as a message signed by a private key can hold the key owner liable for the contents.

###### Secure Communication Architecture

###### The required components for Secure Communication Architecture

* *Network topology design:* A network topology represents the con- nectivity structure among nodes, which can have an impact on the network’s robustness against attacks.
* *Secure routing protocol:* A routing protocol on a network is used to build logical connectivity among nodes; in turn, one of the simplest ways to prevent communication is by attacking the routing proto- col. Thus, we need to consider the security of a routing protocol running on top of a network topology.
* *Secure forwarding:* An adversary who controls a router can alter, drop, and delay existing data packets or inject new packets. Thus, securing individual routers and detecting malicious behaviors are required steps to achieve secure forwarding.
* *End-to-end communication:* From an end-to-end perspective, secrecy and authenticity of data are the most crucial properties. Numerous protocols exist (e.g., SSL/TLS, IPsec, SSH), some low- power devices may need lightweight protocols to perform the associated cryptography.
* *Secure broadcasting:* Many CPS rely on broadcast communication. Especially for dissemination of sensor data, authenticity of the information is important, because an adversary could inject bogus information to cause undesired consequences.
* *DoS defense:* Even when all of the previously mentioned mecha- nisms are in place, an adversary may still be able to prevent communication by mounting a DoS attack.
  + *Jamming defense:* To prevent an external adversary from jamming the wireless network, jamming detection mechanisms can be used to detect attacks and raise alarms. A multitude of methods to coun- ter jamming attacks has been developed, enabling operation during jamming.

###### System and Device Security

* An important area to address to ensure CPS security is vulnerabilities that enable exploitation through software-based attacks, in which either an adversary exploits a software vulnerability to inject malicious code into a system or a malicious insider uses administrative privileges to install and execute malicious code.
* A promising new approach to provide remote code verification is the technology called attestation.
* Code attestation enables an external entity to query the software that is executing on a system in a way that prevents malware from hiding.
* Since attestation reveals a signature of executing code, even unknown malware will alter that signature and, therefore, can be detected.

#### [Advanced Techniques](#_bookmark3)

New and currently developing security schemes need to be aware of new vulnerabilities stemming from both the cyber and physical sides of a CPS and take advantage of both sides.

##### [System Theoretic Approaches](#_bookmark3)

System theoretic approaches take into account the physical properties of the system as a basis for creating new protection mechanisms.

###### Security Requirements

The resiliency of CPS in real- time satisfies the following general properties:

* The CPS should withstand a prespecified list of contingencies.
* The CPS should be able to detect and isolate faults and attacks.
* The performance of the CPS should degrade gracefully with respect to failures or attacks.

A thorough **contingency analysis** can be executed to confirm that the system will continue to operate under constricted conditions.

**Graceful performance degradation** can be effectuated by utilizing a robust control mechanism, which will prevent the system from becoming unstable.

**Fault detection** **and isolation** can be employed to pinpoint and diagnose system faults, as well as some forms of malicious attacks. Countermeasures explicitly designed for distinct attacks can be applied for critical systems.

###### System and Attack Model

Linear time-invariant (LTI) state space model for CPS is used to describe the system.

The LTI model can serve as a good approxi mation of the real system around any operating point. To be specific, for continuous-time systems, the following relationship holds:

*d x*

 *Ax*

* *Bu*
* *BaUa*  *w*

*dt t t t t t*

Here *t*  0 is the index of time, and *xt* and *u* are the state vector

and input vector at time *t*, respectively. The attacker’s input at time

*t* is denoted by *uq*. *w* is white Gaussian noise, which models the

*t*

*t*

uncertainty of the system. *A* and *B* are called the system matrix and the input matrix, respectively.

Finally, the attacker’s input matrix *Ba* is used to model the possible direction of the attacker’s influence on the system.

We further assume that the sensors follow a linear model, which is given by the following expression:

*y*  *Cx*  *ua*  *v*

*t t t t*

Here *yt* is the sensors’ measurements at time *t*, and *v* is white

Gaussian noise, which models the uncertainty in the sensors. *C*

is called the output matrix, and   *diag*( 1 , ,  *m* ) is a diagonal matrix, where  *i* is a binary variable that indicates whether the attacker can change the *i*th measurements:  *i*  1 if the attacker can modify the *i*th measurements; otherwise,  *i*  0.

Given the measurements *yt*, a controller is used to generate the con-

trol input *ut* to stabilize the system or improve its performance. A typi- cal state space controller consists of two parts:

* + A state estimator, which generates an estimate *x*ˆ*t* of the current state

*xt* based on the current and previous measurements *yt*

* + A feedback controller, which generates the control input *ut* based on the state estimate *x*ˆ*t*

In practice, a fixed-gain state estimator and a fixed-gain feedback control- ler are commonly used, which are given by the following expressions:

*d x*ˆ*t*  *Ax*ˆ*t*  *But*  *K* (*yt*  *Cx*ˆ*t* )

*dt*

*ut*  *Lx*ˆ*t*

The matrices *K* and *L* are called the estimation and the control gain, respectively. The closed-loop CPS is stable (in the absence of the attack- er’s input) if and only if all of the eigenvalues of the matrices *A*  *KC* and *A*  *BL* are on the left half-plane (i.e., the real parts of all the eigen- values are strictly negative).

Figure illustrates a simple control system view of the CPS.

The closed-loop CPS is stable (in the absence of the attacker’s input) if and only if all of the eigenvalues of the matrices *A*  *KCA* and *A*  *BL* are located inside the unit circle (i.e., the absolute value of all the eigen- values is strictly less than 1).

Physical

Sensors

Physical System

Actuators

Cyber

Control Logic

**Figure:** *Control system view of a CPS*

The continuous-time and discrete-time models are dynamical mod- els in the sense that the current state *xt* (i.e., *xk*) affects the future state. For some large-scale systems (e.g., power grids), a static model is com- monly adopted, which is given by the following expression:

*y*  *Cx*  *ua*  *v*

*k k k k*

Comparing this equation to the dynamic model, the state *xt* at time *k* is assumed to be unknown and independent from the previous state. A fixed-gain estimator is thus given by the following equation:

*x*ˆ*k*  *Kyk*

###### Characterization of Adversary Models

The a priori knowledge that the attacker has about the system comprises the adversary’s knowledge of the system’s static parameters, such as the matrices *A*, *B*, *C*, the control and estimation gain *L*, *K*, and the statistics of the noise.

By comparison, the disclosure power enables the attacker to obtain real-time system information, such as the current state *xk*, the sensor measurements *yk* , the estimated state *x*ˆ*k*, and the control input *uk*. Lastly, the disruption power consists of the adversary’s capability to dis- turb the system by injecting a malicious control input or by compromis- ing the data integrity of the sensory information or control commands.

Figure illustrates four different attacks based on system knowl- edge, disclosure power, and disruption power. For a pure eavesdropping

System Knowledge

Zero-Dynamics Attack

Eavesdropping

Disclosure Power

DoS Attack

Replay Attack

Disruption Power

**Figure:** *A characterization of cyber-physical attacks*

attack, only disclosure power is needed; for a DoS attack, only disrup- tion power is required.

###### Countermeasures

Advanced countermeasures that involve both cyber and physical elements are key to address the new forms of attacks in CPS. In the following discus- sion, we present a number of the most important innovations in this area.

###### Contingency Analysis

Contingency analysis checks whether the steady-state system is out- side the operating region for each contingency [Shahidehpour05]. The most used security criterion is generally referred to as the *N* − 1 crite- rion. The stipulations of this criterion, among others, are as follows:

* + The system must adhere to constraints on the operating conditions, during normal operation when all (*N*) elements of the system are in operation.
  + For any incident leading to the disconnection of one and only one element of the system, the operating point stays within the required parameters.

The **first stipulation** is tested for different normal states—for example, for peak and off-peak loads in the case of power grids.

The **second stipulation** means that a system component may fail without overloading other components or without violating operating parameters—that is, the intention is that the system will remain in a state where any element may fail but the other elements will remain below their operating limits.

This kind of analysis, which is based on a large number of contin- gent load-flow calculations, is called contingency analysis. Contingency analysis is performed in utilities control centers on a regular basis.

While this security benchmark can be easily generalized consider- ing the loss of more than one element, changing the criterion from *N* − 1 to *N* − 2 and further to *N* − *k*, several factors can rule out such analyses [Zima06]:

* + - Due to the combinatorial nature of the elements, each successive ele- ment failure increases the possible number of contingencies exponen- tially..
    - If the system need to comply with an *N* − *k* criterion, the utilization of the system resources in normal (*N*) operation would be very low, which would result in a bigger operating investment for the same conditions.

###### Fault Detection and Isolation

* + - Fault detection and isolation (FDI) is used in CPS to detect the presence of a fault and pinpoint its type and location.
    - If the distribution of {*yt*} differs from the nominal distribution, then some faults (or possibly attacks) are presented in the CPS.
  + For some systems, however, it is possible for an adversary to carefully
  + construct a sequence of inputs {*ua*}, such that the resulting measurements
    - {*yt*} follow the same distribution as the measurements {*yt*} under normal operation. Such an attack strategy is called a zero-dynamics attack.
    - Since the compromised and normal {*yt*} are statistically identical, no detector can distinguish the compromised system from the normal system.
    - As a result, the zero-dynamics attacks effectively render all FDI schemes useless.
    - To launch a zero-dynamics attack, the adversary needs accurate system knowledge and disruption power to inject malicious control input and/or modify the sensory information and control commands.
  + The zero-dynamics attacks do not require disclosure power, due to the linear nature of the system model.
    - Since zero-dynamics attacks cannot be detected, it is crucial for the system designer **to carefully choose the parameter** of the system such that no zero-dynamics attacks are possible.
    - The existence of a zero-dynamics attack can be checked by the following algebraic conditions. There exists a zero-dynamics attack on the continuous-time system if and only if there exists *s* *C*, *x* *Cn*, and *u* *Cp*, such that we get the following:

*a*

*sI*  *A* *x*  *Baua*  0,

*Cx*  *ua*  0

* + - The existence of zero-dynamics attacks can also be characterized using topological conditions.
    - For the **static system model**, which has been widely adopted in power systems, a bad data detector such as 2 or the largest normalized residue detector detects the corruption in measurements *yk* by checking the residue vector *rk*  *yk*  *Cx*ˆ*k* .
    - For uncorrupted measurements, the residue vector *r* demonstrates a Gaussian distribution.
    - It is possible that a corrupted measurement vector *yk* might generate the same residue *rk* as a normal *yk* . By exploiting this vulnerability, an adversary can inject a stealthy input *ua* into the measurements to change the state estimate *x*ˆ and fool the bad data detector at the same time.

*k*

*k*

* + - As a result, it is impor tant to design the system to rule out the possibility of a “stealthy” attack.

###### Robust Control

* Robust control deals with uncertainty and disturbance in the control
* systems. In the context of CPS security, the attacker’s input can be mod- eled as a disturbance.

*t*

* Reduce the sensitivity of the state {*xt*} with respect to the attacker’s input {*ua*}, and hence guarantee that the system will not deviate greatly from its normal trajectory.

###### Physical Watermarking and Authentication

* A replay attack, like that employed by the Stuxnet malware, requires a significant amount of disclosure power as well as disruption power.
* The attacker, in this scenario, is assumed to be able to inject an external control input into the system and read all sensor readings and modify them arbitrarily.
* Given these capabilities, the attacker can implement an attack strategy similar to the replay attacks encountered in computer security, whereby he or she gives the system a sequence of desired control input while replaying previously recorded measurements.
* Possible countermeasures to distinguish such an attack are to include some form of physical watermarking in the control inputs, which acts as an authentication signal.
* The key idea behind physical watermarking is to embed a small random signal in the control signal, which is kept secret from the attacker.
* As the signal passes through the communication channels, actuators, physical systems, and sensors, it is transformed according to the models of these components.
* As long as the transformed watermark is detected, all the components on the control loop can be considered to be operating with desired functionalities.
* The problem of computing the optimal watermarking signal in the class of independent and identically distributed (IID) Gaussian processes can be cast as a semi-definite programming problem and hence solved efficiently. A diagram of the watermarking scheme is illustrated in Figure, where *uk* is the control signal generated by the feedback controller and *uk* denotes the random watermark.

Attacker

Sensor

Plant

*yk*



*ua*

Actuator

*k*

record/modify

Delay

*uk* – 1

*x^*

*k*

*u*  *u*

Controller

*k k*

Estimator

Failure Detector

**Figure:** *System diagram of a watermarking scheme*