Formal Methods for Cyber-Physical Systems

Complete Course Notes

Contents

1 Introduction to Cyber-Physical Systems

1.1 Definition and Key Features

Definition 1 (Cyber-Physical System)**.** *A cyber-physical system (CPS) is a collection of computing devices that communicate with one another and interact with the physical world via sensors and actuators.*

Cyber-physical systems integrate computation, networking, and physical processes. Components of a CPS include:

- **Cyber components**: Computing platforms, software, communication networks
- **Physical components**: Mechanical systems, electrical systems, physical processes
- **Interface components**: Sensors (to observe physical world) and actuators (to affect physical world)

1.2 Examples of Cyber-Physical Systems

Cyber-physical systems are found in various domains:

- **Medical systems**: MRI scanners, robotic surgery, pacemakers
- **Transportation**: Autonomous vehicles, traffic control systems
- **Manufacturing**: Industrial robots, production lines
- **Energy**: Smart grids, power plants
- **Aerospace**: Aircraft control systems, drones

1.3 Design Challenges

Designing cyber-physical systems presents numerous challenges:

- **Correctness**: Ensuring system behaves according to specifications under all conditions
- **Reliability**: Maintaining operation in the presence of component failures
- **Real-time constraints**: Meeting timing requirements for interactions with physical world
- **Concurrency**: Managing multiple simultaneous activities
- **Safety and security**: Preventing harm to users and environment

1.4 A Better Way to Find and Fix Bugs

Traditional approaches to debugging often involve extensive testing, which cannot guarantee the absence of errors. Formal methods offer:

- **Mathematical precision**: Rigorous definitions of system behavior
- **Systematic analysis**: Comprehensive checking of all possible behaviors
- **Early error detection**: Finding bugs during design rather than after deployment
- **Automatic code generation**: Creating correct-by-construction implementations

Example 1 (MRI Scanner Bug)**.** *In an MRI scanner (a cyber-physical system), there was a software error where the operator could not move the table up when it was partially inside the scanner—behavior that was counterintuitive for users. Using formal methods, engineers could:*

- *Model the system behavior precisely*
- *Specify the desired property (table should be movable in all positions)*
- *Verify that the property is violated in the current design*
- *Generate correct control software that ensures the property is satisfied*

2 Formal Models for Cyber-Physical Systems

2.1 Fundamentals of Computational Models

2.1.1 Functional vs. Reactive Computation

- **Functional computation**:
  - Maps inputs to outputs (e.g., sorting algorithm)
  - Terminates after producing result
  - Described by mathematical functions
- **Reactive computation**:
  - Ongoing interaction with environment
  - Non-terminating (continuously responsive)
  - Described by sequences of input/output interactions

2.1.2 Sequential vs. Concurrent Computation

- **Sequential computation**:
  - Single sequence of instructions
  - Well-understood model (e.g., Turing machines)
- **Concurrent computation**:
  - Multiple simultaneous activities
  - Interaction between components
  - Two main paradigms: synchronous and asynchronous

2.2 Synchronous Reactive Components

2.2.1 Synchrony Hypothesis

Definition 2 (Synchrony Hypothesis)**.** *The time needed to execute the component's computation is negligible compared to the delay between successive input arrivals. Under this hypothesis, computation is considered to take zero time, and outputs are produced simultaneously with inputs.*

Implications of the synchrony hypothesis:

- Execution of update code takes zero time
- Production of outputs and reception of inputs occurs simultaneously
- When multiple components are composed, all execute synchronously

2.2.2 Formal Definition of Synchronous Reactive Components

Definition 3 (Synchronous Reactive Component)**.** *A synchronous reactive component C is defined by a tuple $(I, O, S, Init, React)$ where:*

- *I is a set of typed input variables, defining set Q_I of possible inputs*
- *O is a set of typed output variables, defining set Q_O of possible outputs*
- *S is a set of typed state variables, defining set Q_S of possible states*
- *Init is initialization code defining set Init of initial states*
- *React is reaction description defining set React of reactions of form $s \xrightarrow{i/o} t$, where s, t are states, i is an input, and o is an output*

Example 2 (Delay Component)**.** *A simple delay component that outputs its previous input:*

- *Input variable: in of type Boolean*
- *Output variable: out of type Boolean*
- *State variable: x of type Boolean, initialized to 0*
- *Reaction description: $out := x$; $x := in$*

This component delays the input by one round.

2.2.3 Executions of Synchronous Reactive Components

Definition 4 (Execution)**.** *Given component $C = (I, O, S, Init, React)$, an execution is a sequence: $s_0 \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \xrightarrow{i_3/o_3} s_3 \ldots$ where:*

- *$s_0 \in Init$ is an initial state*
- *For each round $n \geq 1$, $s_{n-1} \xrightarrow{i_n/o_n} s_n \in React$*

2.2.4 Event-based Communication

Components can communicate using events:

- An event can be absent ($\perp$) or present with a value
- event(T) is a type for event variables with values of type T
- x? tests if event x is present
- $x!v$ assigns value v to event x (making it present)
- If no value is assigned to an output event, it is absent by default

Example 3 (Second-To-Minute)**.** *A component that emits an output event every 60th time the input event is present:*

```
event second
int x := 0
if second? then {
    x := x + 1;
    if x == 60 then {
        minute!;
        x := 0
    }
}
event minute
```

2.3 Composition of Components

2.3.1 Compatibility of Components

Definition 5 (Task Graph). *For a synchronous reactive component with input variables I, output variables O, state variables S, and local variables L, a task graph consists of:*

- *A set of tasks, where each task A is specified by:*
  - *Read-set $R_A \subseteq I \cup S \cup O \cup L$*
  - *Write-set $W_A \subseteq S \cup O \cup L$*
  - *Update code defining $\mathit{Update} \subseteq Q_{R_A} \times Q_{W_A}$*
- *A precedence relation $\prec$ over tasks*

Requirements for well-formed task graphs:

1. The precedence relation $\prec$ must be acyclic
2. Each output variable is in the write-set of exactly one task
3. Output/local variables are written before being read
4. Tasks with write conflicts must be ordered

Definition 6 (Interface). *The interface of a component C consists of:*

- *Input variables I*
- *Output variables O*
- *Await dependencies $\succ$, where $y \succ x$ means output y awaits input x*

Definition 7 (Component Compatibility). *Components C_1 and C_2 with await-dependency relations $\succ_1$ and $\succ_2$ are compatible if:*

- *Sets O_1 and O_2 are disjoint (no common outputs)*
- *The relation $(\succ_1 \cup \succ_2)$ of combined await-dependencies is acyclic*

2.3.2 Parallel Composition

Definition 8 (Parallel Composition). *Given compatible components $C_1 = (I_1, O_1, S_1, \mathit{Init}_1, \mathit{React}_1)$ and $C_2 = (I_2, O_2, S_2, \mathit{Init}_2, \mathit{React}_2)$, their parallel composition $C = C_1 \parallel C_2$ is:*

- *Input variables: $(I_1 \cup I_2) \setminus (O_1 \cup O_2)$*
- *Output variables: $O_1 \cup O_2$*

- *State variables:* $S_1 \cup S_2$
- *Initialization:* $Init_1; Init_2$
- *Reaction description: Combined task graph with:*
  - *Tasks:* $\Pi_1 \cup \Pi_2$
  - *Precedence:* $\prec_1 \cup \prec_2 \cup \{(A, A') \mid A \in \Pi_1, A' \in \Pi_2, W_A \cap R_{A'} \neq \emptyset \vee W_{A'} \cap R_A \neq \emptyset\}$

Properties of parallel composition:

- Commutative: $C_1 \parallel C_2 = C_2 \parallel C_1$
- Associative: $(C_1 \parallel C_2) \parallel C_3 = C_1 \parallel (C_2 \parallel C_3)$

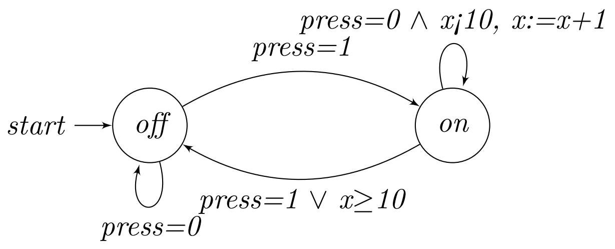
Example 4 (Double Delay)**.** *To create a component that delays input by two rounds:*

1. *Create two instances of Delay: Delay1 and Delay2*
2. *Connect output of Delay1 to input of Delay2 via variable temp*
3. *Compose:* $(Delay1 \parallel Delay2) \setminus temp$

2.4 Extended State Machines

Definition 9 (Extended Finite State Machine)**.** *An Extended Finite State Machine (EFSM) is a synchronous reactive component where:*

- *State variables include a special variable "mode" ranging over a finite set*
- *Reaction description is given by mode switches, each specified by:*
  - *Source mode and target mode*
  - *Guard condition*
  - *Update code for state variables*



Example 5 (Switch Component)**.**
This component has:

- *Input variable: press of type Boolean*

- *State variables: mode $\in \{$off, on$\}$ and x of type integer (initialized to 0)*
- *Four mode switches as shown in the diagram*

3 Safety Requirements and Verification

3.1 Formal Requirements

Definition 10 (Requirement)**.** *A requirement is a desirable property of the executions of a system.*

Types of requirements:

- **Informal**: Stated in natural language or implicit
- **Formal**: Stated explicitly in a mathematically precise manner

Definition 11 (Verification Problem)**.** *Given a requirement ϕ and a system/model C, the verification problem is to prove or disprove that the system C satisfies the requirement ϕ.*

3.2 Safety vs. Liveness

Definition 12 (Safety Requirement)**.** *A safety requirement states that a system always stays within "good" states—nothing bad ever happens.*

Definition 13 (Liveness Requirement)**.** *A liveness requirement states that a system eventually attains its goal—something good eventually happens.*

Key differences:

- Safety violations can be demonstrated by finite executions
- Liveness violations require infinite executions
- Different analysis techniques are needed for each type

3.3 Invariants of Transition Systems

Definition 14 (Transition System)**.** *A transition system T has:*

- *A set S of typed state variables, defining a set Q_S of states*
- *Initialization Init for state variables, defining a set Init $\subseteq Q_S$ of initial states*
- *Transition description Trans, defining a set Trans $\subseteq Q_S \times Q_S$ of transitions*

Definition 15 (Reachable State). *A state s of a transition system is reachable if there is an execution starting in an initial state and ending in s.*

Definition 16 (Invariant). *A property ϕ (a Boolean-valued expression over state variables) is an invariant of transition system T if every reachable state satisfies ϕ.*

Safety verification approach:

- Express the desired safety requirement as a property ϕ over state variables
- Check if ϕ is an invariant of the system
- If ϕ is not an invariant, find a counterexample (execution leading to a state violating ϕ)

3.4 Requirement-Based Design

Requirement-based design is a systematic approach to designing systems:

- Given: Input/output interface of system C, model E of the environment, safety property ϕ
- Design problem: Fill in details of C so that the composite system $C \parallel E$ satisfies the invariant ϕ

Example 6 (Railroad Controller). *Consider a railroad system with two trains approaching a bridge from opposite directions. The bridge can accommodate only one train at a time, and each entrance is equipped with a signal.*

Safety Requirement: Trains should not be on the bridge simultaneously.
Controller Design:

- *Environment model: Train components that can be in states {away, wait, bridge}*
- *Controller interface: Controls signals {red, green} at each entrance*
- *Design task: Develop a controller that ensures the safety property "$\neg(mode_W = bridge \land mode_E = bridge)$"*

3.5 Safety Monitors

Definition 17 (Safety Monitor). *A monitor M for a system observes its inputs/outputs and enters an error state if undesirable behavior is detected. Formally, a monitor is specified as an extended state machine with:*

- *Input variables = input/output variables of the system being monitored*

- *A subset F of modes declared as accepting (error states)*

Safety verification with monitors:

- Undesirable behavior: an execution that leads monitor to an accepting state
- Safety verification: Check whether "(monitor.mode $\notin F$)" is an invariant of System $C \parallel M$

Example 7 (Fairness Monitor for Railroad)**.** *To check that while the west train waits, the east train is not allowed on the bridge twice in succession:*

$$\begin{tikzcd}
\textit{start} \arrow[r] & \textit{0} \arrow[r, "mode_W\ =\ wait"] & \textit{1} \arrow[r, "mode_E\ =\ bridge"] & \textit{2} \arrow[r, "mode_E\ =\ bridge"] & \textit{3}
\end{tikzcd}$$

3.6 Automated Invariant Verification

3.6.1 Decidability of Invariant Verification

Theorem 1. *The invariant verification problem is undecidable in general.*

Theorem 2. *The invariant verification problem for finite-state systems is decidable.*

For finite-state systems:

- If T has k Boolean variables, the total number of states is 2^k
- Verifier can systematically search through all possible states
- Complexity is exponential in time (and polynomial in memory)

3.6.2 On-the-fly Enumerative Search

Properties of DFS Search:

- Correctness: If the algorithm returns $\emptyset$, the property ϕ is not reachable; if it returns a sequence of states, this is a witness execution
- Termination: If the number of reachable states is finite, the algorithm terminates

4 Symbolic Verification Algorithms

4.1 Symbolic State Representation

Definition 18 (Region)**.** *A region over variables X is a data structure that represents a set of states assigning values to X.*

Algorithm 1 On-the-fly DFS Search (Part 1)

```
function REACHABLE($T, \phi$)
    Reach $\leftarrow \emptyset$
    $s \leftarrow$ FirstInitState($T$)
    while $s \neq$ null do
        if $s \notin$ Reach then
            exec $\leftarrow$ DFS($s, \phi$, Reach)
            if exec $\neq \emptyset$ then
                return exec
            end if
        end if
        $s \leftarrow$ NextInitState($T, s$)
    end while
    return $\emptyset$
end function
```

Algorithm 2 On-the-fly DFS Search (Part 2)

```
function DFS($s, \phi$, Reach)
    Reach $\leftarrow$ Reach $\cup \{s\}$
    if Satisfies($s, \phi$) then
        return List($s$)
    end if
    $t \leftarrow$ FirstSuccState($T, s$)
    while $t \neq$ null do
        if $t \notin$ Reach then
            exec $\leftarrow$ DFS($t, \phi$, Reach)
            if exec $\neq \emptyset$ then
                return Append($s$, exec)
            end if
        end if
        $t \leftarrow$ NextSuccState($T, s, t$)
    end while
    return $\emptyset$
end function
```

Symbolic representation of a transition system T with state variables S:

- Region ϕ_I over S for initial states
- Region ϕ_T over $S \cup S'$ for transitions, where S' represents primed variables (values in the target state)

Basic operations on regions:

- Union(A, B): Returns region containing states in either A or B
- Intersect(A, B): Returns region containing states in both A and B
- Diff(A, B): Returns region containing states in A but not in B
- IsEmpty(A): Returns True if region A contains no states
- Exists(A, X): Returns projection of A by quantifying variables in X
- Rename(A, X, Y): Renames variables in X to corresponding variables in Y

4.1.1 Image Computation

Definition 19 (Post-Image)**.** *Given a region A, the post-image Post(A) is the set of successors of states in A: Post$(A) = \{t \mid$ there exists a state $s \in A$ and a transition $(s, t)\}$*

Symbolic computation of post-image:

$$\text{Post}(A, \text{Trans}) = \text{Rename}(\text{Exists}(\text{Intersect}(A, \text{Trans}), S), S', S) \tag{1}$$

Steps:

1. Take intersection of A and Trans
2. Project out variables in S using existential quantification
3. Rename primed variables to get a region over S

4.1.2 Pre-Image Computation

Definition 20 (Pre-Image)**.** *Given a region A, the pre-image Pre(A) is the set of predecessors of states in A: Pre$(A) = \{s \mid$ there exists a state $t \in A$ and a transition $(s, t)\}$*

Symbolic computation of pre-image:

$$\text{Pre}(A, \text{Trans}) = \text{Exists}(\text{Intersect}(\text{Rename}(A, S, S'), \text{Trans}), S') \tag{2}$$

Steps:

1. Rename S to primed variables to get a region over S'
2. Take intersection with Trans
3. Project out variables in S' using existential quantification

4.2 Binary Decision Diagrams

Definition 21 (Binary Decision Diagram)**.** *A Binary Decision Diagram (BDD) is a data structure for representing Boolean functions as directed acyclic graphs.*

Definition 22 (Reduced Ordered Binary Decision Diagram)**.** *A Reduced Ordered Binary Decision Diagram (ROBDD) is a BDD where:*

- *Variables appear in the same order on each path*
- *Isomorphic subgraphs are merged*
- *Redundant nodes (where low and high children are identical) are eliminated*

Key properties of ROBDDs:

- Canonical: For a given variable ordering, each Boolean function has a unique ROBDD
- Minimal: Smallest possible decision graph given the ordering restriction
- Efficient operations for Boolean operations (AND, OR, NOT)
- Efficient satisfiability and validity checking

ROBDD operations:

- True(): Returns the ROBDD for constant 1
- False(): Returns the ROBDD for constant 0
- Var(x): Returns the ROBDD for the formula x
- Not(B): Returns the ROBDD for $\neg f(B)$
- And(B_1, B_2): Returns the ROBDD for $f(B_1) \wedge f(B_2)$
- Or(B_1, B_2): Returns the ROBDD for $f(B_1) \vee f(B_2)$
- Exists(B, X): Returns the ROBDD for $\exists X.f(B)$

4.3 Symbolic Reachability Analysis

Properties of symbolic search:

- Correctness: When the algorithm stops, its answer (whether property ϕ is reachable) is correct
- Termination: Number of iterations depends on the length of shortest execution leading to a state satisfying ϕ or the diameter of the state space
- Efficiency depends on the symbolic representation (e.g., ROBDDs)

Algorithm 3 Symbolic Breadth-First-Search Algorithm

```
1: function SymbolicReachable(Init, Trans, $\phi$)
2:     Reach ← Init
3:     New ← Init
4:     while not IsEmpty(New) do
5:         if not IsEmpty(Intersect(New, $\phi$)) then
6:             return True
7:         end if
8:         New ← Diff(Post(New, Trans), Reach)
9:         Reach ← Union(Reach, New)
10:    end while
11:    return False
12: end function
```

4.4 Witness Generation

To modify the symbolic breadth-first search algorithm to generate witnesses:
Additional operations required:

- PickState(A): Returns a state contained in region A
- Pre(s, Trans): Computes set of predecessors of state s

5 Liveness Requirements and Verification

5.1 Temporal Logic

Definition 23 (Linear Temporal Logic (LTL))**.** *Linear Temporal Logic is a formal language for specifying properties of infinite sequences (traces) of states.*

Basic components:

- Base formulas: Boolean-valued expressions over state variables
- Logical connectives: $\wedge$, $\vee$, $\neg$, $\rightarrow$
- Temporal operators: $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next), $\mathcal{U}$ (until)

5.1.1 LTL Semantics

Let $\rho = q_1, q_2, q_3, \ldots$ be a trace (infinite sequence of valuations).

- $\rho \models \phi$ for a base formula ϕ if $q_1 \models \phi$
- $\rho \models \Box\phi$ if for all $j \geq 1$, $(q_j, q_{j+1}, q_{j+2}, \ldots) \models \phi$

Algorithm 4 Symbolic Breadth-First-Search with Witness Generation

```
function WitnessReachable(Init, Trans, $\phi$)
    Reach ← Init
    $\text{New}_1$ ← Init
    $k \leftarrow 1$
    while not IsEmpty($\text{New}_k$) do
        if not IsEmpty(Intersect($\text{New}_k, \phi$)) then
            $s \leftarrow$ PickState(Intersect($\text{New}_k, \phi$))
            path ← [$s$]                              ▷ path is a list of states
            for $i \in \{k-1, \dots, 1\}$ do
                Pred ← Intersect(Pre($s$, Trans), $\text{New}_i$)
                $s \leftarrow$ PickState(Pred)
                append $s$ to the head of path
            end for
            return path
        end if
        $\text{New}_{k+1}$ ← Diff(Post($\text{New}_k$, Trans), Reach)
        Reach ← Union(Reach, $\text{New}_{k+1}$)
        $k \leftarrow k + 1$
    end while
    return [ ]                    ▷ $\phi$ not reachable: return empty path
end function
```

- $\rho \models \Diamond\phi$ if for some $j \geq 1$, $(q_j, q_{j+1}, q_{j+2}, \ldots) \models \phi$
- $\rho \models \bigcirc\phi$ if $(q_2, q_3, q_4, \ldots) \models \phi$
- $\rho \models \phi\mathcal{U}\psi$ if for some $j \geq 1$, $(q_j, q_{j+1}, q_{j+2}, \ldots) \models \psi$ and for all $i < j$, $(q_i, q_{i+1}, q_{i+2}, \ldots) \models \phi$

5.1.2 Derived Operators

- Repeatedly $\phi = \Box\Diamond\phi$: ϕ holds infinitely often
- Persistently $\phi = \Diamond\Box\phi$: ϕ eventually holds forever

Example 8 (Railroad Controller Requirements in LTL). • *Safety: Trains should not be on bridge simultaneously* $\Box\neg(\mathit{mode}_W = \mathit{bridge} \wedge \mathit{mode}_E = \mathit{bridge})$

- *Liveness: A waiting west train is eventually allowed to enter* $\Box(\mathit{mode}_W = \mathit{wait} \to \Diamond(\mathit{signal}_W = \mathit{green}))$
- *Conditional liveness: If east train doesn't stay on bridge forever, west train is allowed to enter when waiting* $\Box\Diamond(\mathit{mode}_E \neq \mathit{bridge}) \to \Box(\mathit{mode}_W = \mathit{wait} \to \Diamond(\mathit{signal}_W = \mathit{green}))$

5.2 Büchi Automata

Definition 24 (Büchi Automaton). *A Büchi automaton $M = \langle V, Q, \mathit{Init}, F, E\rangle$ consists of:*

- *V: set of Boolean input variables*
- *Q: finite set of states*
- *$\mathit{Init} \subseteq Q$: set of initial states*
- *$F \subseteq Q$: set of accepting states*
- *E: set of edges, where each edge is of the form $q \xrightarrow{\mathit{Guard}} q'$ with Guard being a Boolean condition over V*

Definition 25 (Accepting Run). *Given an input trace $\rho = v_1, v_2, v_3, \ldots$ over V, an accepting run of M over ρ is an infinite sequence of states $q_0, q_1, q_2, \ldots$ such that:*

- *$q_0 \in \mathit{Init}$*
- *For each i, there exists an edge $q_i \xrightarrow{\mathit{Guard}} q_{i+1}$ such that input v_i satisfies Guard*

- *There are infinitely many positions i such that $q_i \in F$*

Definition 26 (Generalized Büchi Automaton)**.** *A generalized Büchi automaton has k accepting sets $F_1, F_2, \ldots, F_k$. An execution is accepting if for each j, some state in F_j appears infinitely often.*

5.3 From LTL to Büchi Automata

Theorem 3. *For any LTL formula ϕ, there exists a Büchi automaton M_ϕ that accepts exactly those traces that satisfy ϕ.*

The tableau construction algorithm converts an LTL formula to a Büchi automaton:

1. Define Sub(ϕ), the closure of formula ϕ, containing:
    - All syntactic subformulas of ϕ
    - For each subformula ψ, also include $\neg\psi$
    - For temporal operators, include their "next-time" parts (e.g., for $\Box\psi$, include $\bigcirc\Box\psi$)
2. Define states of the automaton as consistent subsets of Sub(ϕ) satisfying:
    - For every $\psi \in$ Sub(ϕ), either ψ or $\neg\psi$ is in the state
    - States satisfy propositional logic rules (e.g., $\psi_1 \wedge \psi_2$ is in the state iff both ψ_1 and ψ_2 are)
    - States satisfy rules for temporal operators based on their inductive definitions
3. Define transitions between states based on next-formulas
4. Define accepting conditions to ensure eventualities are satisfied

5.4 Repeatability Checking

Definition 27 (Repeatable Property)**.** *Given a transition system and a set F of states, F is repeatable if there exists an infinite execution that visits states in F infinitely often.*

To check whether a system C satisfies an LTL formula ϕ:

1. Construct the Büchi automaton $M_{\neg\phi}$ for the negation of ϕ
2. Check if the set of accepting states is repeatable in the composition of C and $M_{\neg\phi}$

Finding witnesses for repeatability:

Algorithm 5 Symbolic Repeatability Algorithm

```
function SymbolicRepeatable(Init, F, Trans)
                                                  ▷ Phase 1: compute Reach
    Reach ← Init
    New ← Init
    while not IsEmpty(New) do
        New ← Diff(Post(New, Trans), Reach)
        Reach ← Union(Reach, New)
    end while
                                                  ▷ Phase 2: check repeatability
    Recur ← Intersect(Reach, $F$)
    while not IsEmpty(Recur) do
        PreReach ← $\emptyset$
        New ← Pre(Recur, Trans)
        while not IsEmpty(New) do
            PreReach ← Union(PreReach, New)
            if IsSubset(Recur, PreReach) then
                return True
            end if
            New ← Diff(Pre(New, Trans), PreReach)
        end while
        Recur ← Intersect(Recur, PreReach)
    end while
    return False
end function
```

1. Find a state s in the final Recur set
2. Compute set $R = \{t \in \text{Recur} \mid t \text{ is reachable from } s \text{ with path } \geq 1\}$
3. If $s \in R$, build a loop from s to s
4. Connect this loop to an initial state to form a lasso-shaped witness

6 Controller Synthesis for Discrete-Event Systems

6.1 Discrete-Event Systems

Definition 28 (Discrete-Event System)**.** *A discrete-event system is a model where:*

- *Time is discrete*
- *At every step, exactly one event happens*
- *Components communicate by sharing events*

Definition 29 (Automaton)**.** *An automaton $M = \langle Q, E, \rightarrow, q_0, F \rangle$ consists of:*

- *Finite set Q of states*
- *Finite set E of events*
- *Transition relation $\rightarrow \subseteq Q \times E \times Q$*
- *Initial state $q_0 \in Q$*
- *Set $F \subseteq Q$ of marked (accepting) states*

Definition 30 (Language of an Automaton)**.** *For an automaton M:*

- *Generated language: $L(M) = \{w \in E^* \mid q_0 \xrightarrow{w}\}$*
- *Marked language: $L_m(M) = \{w \in E^* \mid q_0 \xrightarrow{w} q', q' \in F\}$*

Properties of automata:

- Deadlock: a state with no outgoing transitions
- Blocking: a state that cannot reach a marked state
- Non-blocking: every reachable state can reach a marked state

6.2 Supervisory Control Theory

Definition 31 (Plant and Supervisor)**.** *In supervisory control:*

- *The plant P is an automaton representing the physical system*
- *The supervisor S is an automaton that controls the plant by enabling/disabling events*
- *Events are partitioned into controllable events E_c and uncontrollable events E_u*

Definition 32 (Controllability)**.** *A supervisor S is controllable for a plant P with set of uncontrollable events E_u if whenever $w \in L(P \parallel S)$ and $wu \in L(P)$ for some $u \in E_u$, then also $wu \in L(P \parallel S)$.*

Definition 33 (Proper Supervisor)**.** *An automaton S is a proper supervisor for a plant P if:*

- *$P \parallel S$ is non-blocking*
- *S is controllable for plant P*

Definition 34 (Maximal Permissiveness)**.** *Given a plant P, a proper supervisor S is maximally permissive if for each proper supervisor S', it holds that $L_m(P \parallel S') \subseteq L_m(P \parallel S)$.*

6.3 Controller Synthesis Algorithm

Basic supervisory control problem:

- Given a plant P with set of uncontrollable events E_u
- Find a maximally permissive proper supervisor S for P

Algorithm steps:

1. Start with the uncontrolled plant as a candidate supervisor
2. Repeat until no more blocking states are found:
   (a) Compute the set of blocking states
   (b) Compute the set of bad states (blocking states and states that can reach a bad state via uncontrollable events)
   (c) Remove transitions with controllable events that target bad states
   (d) Remove unreachable states and transitions

Theorem 4. *The supervisor S produced by the synthesis algorithm is:*

- *A proper supervisor for plant P*
- *Maximally permissive*

6.4 Extended Finite Automata for Controller Synthesis

Definition 35 (Extended Finite Automaton)**.** *An extended finite automaton extends a standard automaton with:*

- *A set of discrete variables with finite domains*
- *Guards on transitions (predicates over variables)*
- *Updates on transitions (assignments to variables)*

Controller synthesis with extended finite automata:

1. Define plant and requirements as extended finite automata
2. Create parallel composition of plant and requirements
3. Compute non-blocking conditions symbolically
4. Compute bad state conditions (states that lead to controllability issues)
5. Tighten guards on controllable transitions to avoid bad states

State-based requirements:

- Event conditions: $\{e_1, \ldots, e_n\} \Rightarrow \text{Pred}$ (events can only occur when predicate is satisfied)
- Invariants: Predicates that must hold in all states

7 Case Studies

7.1 Machine-Warehouse Example

Example 9 (Machine-Warehouse System)**.** *A system consisting of:*

- *Machine: processes workpieces, has states IDLE and BUSY*
- *Warehouse: stores finished workpieces, has states EMPTY, HALF, and FULL*
- *Events: start (machine starts processing), finish (machine finishes and workpiece goes to warehouse), remove (warehouse removes a workpiece)*

Requirements:

1. *The warehouse stores at most one workpiece*

Synthesis steps:

1. *Model machine and warehouse as automata*
2. *Create the uncontrolled plant by parallel composition*
3. *Formalize the requirement as an automaton*
4. *Apply synthesis algorithm to generate a proper supervisor*

The resulting supervisor ensures the machine can only start processing when the warehouse is empty.

7.2 Workcell with an AGV

Example 10 (Workcell with Automated Guided Vehicle). *A workcell consisting of:*

- *Two machines M1 and M2*
- *An automated guided vehicle (AGV)*
- *Events: start_M1, start_M2 (controllable), end_M1, end_M2 (uncontrollable), to_B (AGV takes workpiece to buffer B)*

Requirements:

1. *Ensure safe operation of the system*
2. *Prevent deadlocks*

Synthesis results in a supervisor that controls the start operations of machines to ensure proper flow of workpieces.

7.3 Manufacturing Process Requirements

Example 11 (Manufacturing Process). *A manufacturing system with:*

- *Multiple machines that can be IDLE, ACTIVE, or DOWN*
- *A buffer that can be EMPTY or FULL*
- *Events for machine operations and state changes*

Requirements expressed using event conditions and invariants:

1. *Machine 1 can start processing only if the buffer is empty*
2. *Machine 2 can start processing only if the buffer is full*
3. *Machine 1 cannot start if Machine 2 is down*
4. *If both machines are down, Machine 2 is repaired before Machine 1*

The requirements are formalized using extended finite automata and state-based specifications.

8 Conclusion

8.1 Summary of Key Concepts

This course has covered the following key areas of formal methods for cyber-physical systems:

- **Formal Models**: Synchronous reactive components, extended state machines, discrete-event systems
- **Safety Verification**: Invariants, reachability analysis, monitors
- **Symbolic Algorithms**: BDDs, symbolic state representation, image computation
- **Liveness Verification**: Temporal logic, Büchi automata, repeatability checking
- **Controller Synthesis**: Supervisory control theory, extended finite automata for synthesis

8.2 Applications of Formal Methods

Formal methods provide significant benefits in the design of cyber-physical systems:

- Early detection of design flaws
- Systematic exploration of all possible behaviors
- Automatic generation of correct-by-construction controllers
- Clear documentation of system requirements and properties
- Increased confidence in safety-critical systems

8.3 Future Directions

Active research areas in formal methods for cyber-physical systems include:

- Scalability of verification techniques
- Integration with machine learning
- Formal methods for distributed and adaptive systems
- Human-in-the-loop cyber-physical systems
- Security verification for cyber-physical systems