Table of Contents

Synchronous Reactive Components

Weekly Exercise 1.1

> Design a reactive component with three Boolean input variables x, y and reset, and a Boolean output variable z. The desired behaviour is the following. The component waits until is has encountered a round in which the input variable x is high (1) and a round in which the input variable y is high (1), and as soon as both of these have been encountered it sets the output z to high. It repeats the behaviour when, in a subsequent round, the input variable reset is high. By default the output z is low (0).

Solution: We can model this using an extended state machine with four states:

- `initial` : Neither x nor y has been high yet
- `x_seen` : x has been high, but not y
- `y_seen` : y has been high, but not x
- `both_seen` : Both x and y have been high

```
// Extended State Machine Notation
// State variables
enum mode := {initial, x_seen, y_seen, both_seen} mode := initial;
bool z := 0;

// Mode switches
initial → x_seen    : x=1 & y=0 & reset=0
initial → y_seen    : x=0 & y=1 & reset=0
initial → both_seen : x=1 & y=1 & reset=0, z:=1

x_seen → x_seen    : x=0 & y=0 & reset=0
```

```
x_seen → both_seen : (x=0 | x=1) & y=1 & reset=0, z:=1
x_seen → initial   : reset=1, z:=0

y_seen → y_seen    : x=0 & y=0 & reset=0
y_seen → both_seen : x=1 & (y=0 | y=1) & reset=0, z:=1
y_seen → initial   : reset=1, z:=0

both_seen → both_seen : reset=0
both_seen → initial   : reset=1, z:=0
```

Weekly Exercise 1.2

> Design a synchronous reactive component CountPositive with an integer input variable x, an input event variable reset, and a natural output variable y, with the following behaviour. In the first round, the output y is 0. For every subsequent round i, let j < i the most recent round before round i in which the event reset is present (if reset is absent in all rounds before i, then j = 0): the output y should be the number of rounds in which the variable y is positive (greater than 0) in rounds j, j + 1 up to i − 1.

Solution:

```
// Component definition
event reset;
int x;
nat y;

// State variables
nat count := 0;  // Counts positive inputs since last reset

// Reaction code
if reset? then {
  count := 0;
  y := 0;
} else {
  if x > 0 then {
    count := count + 1;
  }
  y := count;
}
```

This component is deterministic as it has a single possible reaction for each input. It is not finite-state because the `count` variable can grow unboundedly.

Weekly Exercise 1.3

> Consider the component Counter of the figure below. The component is not input-enabled: why? Modify the component to make it input-enabled.

Solution: The component is not input-enabled because when `x = 0` and `dec = 1`, there is no reaction defined (it tries to decrement a counter that's already at 0).

Modified component to make it input-enabled:

```
nat x := 0;
nat out;

if (inc = 1 && dec = 0) then {
    x := x + 1;
    out := x;
} else if (inc = 0 && dec = 1 && x > 0) then {
    x := x - 1;
    out := x;
} else if (inc = 0 && dec = 0) then {
    out := x;
} else if (inc = 1 && dec = 1) then {
    out := x;  // Conflicting inputs, keep value
} else if (inc = 0 && dec = 1 && x = 0) then {
    out := x;  // Cannot decrement at 0, keep value
}
```

Exercise 1 (ex-1001.pdf)

> What does this component do? Describe in words the behaviour of the component.

The component referred to is likely the "Delay" component from other exercises. This component takes a Boolean input and outputs the previous value of the input. The output in the first round is 0 (the initial value of the state variable).

Exercise 2 (ex-1001.pdf)

> Desired behaviour: Issue the output event out every 60th time the input event is present. Write the reaction code of the component.

Solution:

```
event second;
event minute;
int x := 0;
```

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

Composition of Components

Exercise 1 (ex-1006.pdf)

> Consider the feedback compositions Delay ‖ Inverter...

Solution: a. Input variables: ∅ (empty set) - Both inputs are connected internally b. Output variables: `{out}` c. State variables: `{x}` d. Initialization: `x := 0` e. Reaction code:

```
out := x;
x := ~out;
```

Exercise 2 (ex-1006.pdf)

> Consider the feedback compositions Relay ‖ Inverter...

Solution: a. Input variables: ∅ (empty set) b. Output variables: `{out}` c. State variables: ∅ (empty set) d. Initialization: None (no state variables) e. Reaction code:

```
out := ~out;
```

This creates an oscillator that toggles the output value in each round.

Task Graphs and Await Dependencies

Exercise 1 (ex-1011.pdf)

> Consider the synchronous reactive component shown in Figure 1. List all possible reactions of the component. Does the output y await x? Does the output z await x?

Solution: Reactions:

- In task A_1: The output `y` is set to the value of `u`
- In task A_2: State variable `u` is set to `¬x` and output `z` is set based on `choose(x, u)`

The output `y` does not await `x` because task A_1 does not read `x`. The output `z` awaits `x` because task A_2 reads `x` before writing to `z`.

Exercise 2 (ex-1011.pdf)

> Consider a synchronous reactive component C with an input variable x, and output variables y and z. The component has two tasks, A_1 and A_2, such that the output y belongs to the write-set of the task A_1, and the output z belongs to the write-set of the task A_2. If we know that the output y awaits the input x, but the output z does not await x, then what can we conclude regarding the precedence constraints between the tasks A_1 and A_2?

Solution: If the output `y` awaits input `x`, then task A_1 must read `x` in its read-set. If the output `z` does not await input `x`, then task A_2 must not read `x`.

Since task A_1 reads `x` and task A_2 writes to `z` without reading `x`, there cannot be a precedence constraint $A_2 \prec A_1$ (otherwise `z` would await `x` transitively).

There might be a constraint $A_1 \prec A_2$ if A_2 reads any variable written by A_1, but this is not necessarily the case from the given information.

Exercise 1 (ex-1012.pdf)

> List the await dependencies of the components in Figure 1.

Solution: For component C:

- Output `y` awaits input `u`, which doesn't await anything (initialized to 0)
- Output `z` awaits input `x`
- Output `u` awaits input `x`

For component D_1:

- Output `x` awaits inputs `y` and `z` directly

For component D_2:

- Output `x` awaits state variable `t`, which in turn awaits inputs `y` and `z`
- Therefore, `x` awaits `y` and `z`

Exercise 2 (ex-1012.pdf)

> Are the components C and D_1 compatible? If they are compatible, describe the input-output behaviour of the composition $(C \| D_1) \setminus \{y, z\}$.

Solution: To determine compatibility, we need to check if:

1. The output sets are disjoint

2. The combined await-dependency relation is acyclic

Output sets: C outputs {y, z, u} and D_1 outputs {x}. These are disjoint.

Await dependencies:

- In C: `y` awaits nothing, `z` awaits `x`, `u` awaits `x`
- In D_1: `x` awaits `y` and `z`

This creates a cycle: `x` awaits `y` and `z`, while `z` awaits `x`.

Therefore, C and D_1 are not compatible due to a cyclic await-dependency relation.

Exercise 3 (ex-1012.pdf)

> Are the components C and D_2 compatible? If they are compatible, describe the input-output behaviour of the composition $(C \| D_2) \setminus \{y, z\}$.

Solution: Output sets: C outputs {y, z, u} and D_2 outputs {x, t}. These are disjoint.

Await dependencies:

- In C: `y` awaits nothing, `z` awaits `x`, `u` awaits `x`
- In D_2: `t` awaits `y` and `z`, `x` awaits `t`

There is no cycle in the await dependencies. Therefore, C and D_2 are compatible.

The behavior of $(C \| D_2) \setminus \{y, z\}$ would have:

- Input variables: none (all are connected internally)
- Output variables: {u, x}
- State variables: {u, t}

In each round, component C computes `y` from `u`, then D_2 computes `t` from `y` and `z`. Next, C computes `z` and `u` from `x`, and finally D_2 computes `x` from `t`. This creates a feedback loop that stabilizes within each round.

Exercise 4 (ex-1012.pdf)

> Define a variant C' of component C such that C' and D_1 are compatible and $(C' \| D_1) \setminus \{y, z\}$ is equivalent to $(C \| D_2) \setminus \{y, z\}$ with respect to the input-output behaviour.

Solution: To make C' compatible with D_1, we need to break the cyclic await dependency. We can modify C to:

```
bool u := 0;
bool x;
```

```
bool y;
bool z;

A₁: u → y   (y := u)
A₂: x → z   (z := x)
A₃: → u     (u := ¬x from previous round)
```

This eliminates the await dependency of `u` on `x` in the current round, breaking the cycle.

Exercise 1 (ex-1011.pdf - DoubleSplitDelay)

> Consider the component DoubleSplitDelay defined as (SplitDelay[out 7→ temp]‖SplitDelay[in 7→ temp]) \ temp. This component is similar to the component DoubleDelay except we use instances of the component SplitDelay. Show the "compiled" version of DoubleSplitDelay.

Solution: The compiled version of DoubleSplitDelay has:

State variables: {x_1, x_2} (one from each SplitDelay instance) Input variables: {in} Output variables: {out} Local variables: {temp}

Tasks:

- A_{11}: $x_1 \rightarrow$ temp
- A_{21}: temp $\rightarrow x_2$
- A_{12}: $x_2 \rightarrow$ out
- A_{22}: in $\rightarrow x_1$

Precedence constraints:

- $A_{11} \prec A_{21}$ (temp is read by A_{21} after being written by A_{11})
- $A_{12} \prec A_{22}$ (no dependency, but needed for SplitDelay_2)

Await dependencies:

- temp awaits x_1
- x_2 awaits temp
- out awaits x_2
- x_1 awaits in

By transitivity, out awaits in with a delay of 2 rounds.

Safety Requirements and Verification

Weekly Exercise 3.1

> Given two natural numbers m and n, consider the transition system Mult(m, n) that multiplies the input numbers n, m shown in the figure below. Describe this transition system symbolically using initialization formula φ_i and transition formula φ_t.

Solution: For the Mult(m, n) system:

Initialization formula φ_i:

```
ϕᵢ ≡ (mode = loop) ∧ (x = m) ∧ (y = 0)
```

Transition formula φ_t:

```
ϕₜ ≡ (mode = loop ∧ x = 0 ∧ mode' = stop ∧ x' = x ∧ y' = y) ∨
     (mode = loop ∧ x > 0 ∧ mode' = loop ∧ x' = x−1 ∧ y' = y+n)
```

Weekly Exercise 3.2

> Define a pre-image computation Pre using the symbolic operations.

Solution: The pre-image computation Pre(A, Trans) can be defined as:

```
Pre(A, Trans) = Exists(Intersect(Rename(A, S, S'), Trans), S')
```

Where:

- A is the region for which we want to compute pre-images
- Trans is the transition relation
- S is the set of state variables
- S' is the set of primed state variables
- Rename(A, S, S') renames variables in S to their primed versions in S'
- Intersect computes the intersection of two regions
- Exists performs existential quantification over variables

Weekly Exercise 3.3

> Suppose we want to modify the symbolic breadth-first search algorithm so that when it finds the property φ to be reachable, it outputs a witness execution.

Solution: Additional operations needed:

- `PickState(A)` : Returns a representative state from a non-empty region A
- `Pre(s, Trans)` : Computes the set of predecessors of a specific state s

Modified algorithm:

```
function WitnessReachable(Init, Trans, ϕ)
    Reach ← Init
    New₁ ← Init
    k ← 1
    while not IsEmpty(Newₖ) do
        if not IsEmpty(Intersect(Newₖ, ϕ)) then
            s ← PickState(Intersect(Newₖ, ϕ))
            path ← [s]  // path is a list of states
            for i ∈ {k-1, ..., 1} do
                Pred ← Intersect(Pre(s, Trans), Newᵢ)
                s ← PickState(Pred)
                append s to the front of path
            end for
            return path
        end if
        Newₖ₊₁ ← Diff(Post(Newₖ, Trans), Reach)
        Reach ← Union(Reach, Newₖ₊₁)
        k ← k + 1
    end while
    return [ ]  // ϕ not reachable: return empty path
end function
```

Exercise 1 (ex-1015.pdf)

> The system RailRoadSystem2 = TrainW ‖ TrainE ‖ Controller2 has six state variables and 144 possible states. How many states are reachable?

Solution: To determine the number of reachable states, we need to analyze the constraints imposed by the Controller2 on the composed system:

1. The controller ensures that west and east signals are never both green
2. The controller ensures that a train can only be on the bridge if its signal is green

From these constraints, the states where both trains are on the bridge simultaneously are unreachable. Additionally, states where a train is on the bridge but its signal is red are also unreachable.

The analysis shows that the number of reachable states is 42 out of the total 144 states.

Enumerative Search (ex-1015.pdf)

> Write an algorithm that takes as inputs a transition system T and a property φ. If φ is reachable in T your algorithm should return a witness, that is, an execution of T that starts from an initial state and ends in a state where φ is satisfied.

Solution:

```
function Reachable(T, ϕ)
    Reach ← ∅
    s ← FirstInitState(T)
    while s ≠ null do
        if s ∉ Reach then
            exec ← DFS(s, ϕ, Reach)
            if exec ≠ ∅ then
                return exec
            end if
        end if
        s ← NextInitState(T, s)
    end while
    return ∅
end function

function DFS(s, ϕ, Reach)
    Reach ← Reach ∪ {s}
    if Satisfies(s, ϕ) then
        return List(s)
    end if
    t ← FirstSuccState(T, s)
    while t ≠ null do
        if t ∉ Reach then
            exec ← DFS(t, ϕ, Reach)
            if exec ≠ ∅ then
                return Append(s, exec)
            end if
        end if
        t ← NextSuccState(T, s, t)
    end while
    return ∅
end function
```

Symbolic Verification Algorithms

Weekly Exercise 4.1

> Write the pseudo-code for the function CountFalse(B) that takes a ROBDD B with n variables x_1, ..., x_n, and returns the number of valuations for the n variables making B false.

Solution:

```
function CountFalse(B)
    return CountPaths(B, 0) * 2^NumMissingVars(B)
end function

function CountPaths(B, target)
    if B = target then
        return 1
    else if B = (1-target) then
        return 0
    else
        B_node ← BDDPool[B]
        low_count ← CountPaths(Low(B_node), target)
        high_count ← CountPaths(High(B_node), target)
        return low_count + high_count
    end function

function NumMissingVars(B)
    if B = 0 or B = 1 then
        return n
    else
        B_node ← BDDPool[B]
        vars_seen ← 1  // Current variable
        current_var ← Label(B_node)

        low_missing ← NumMissingVarsRec(Low(B_node), current_var + 1)
        high_missing ← NumMissingVarsRec(High(B_node), current_var + 1)

        return max(low_missing, high_missing)
    end if
end function

function NumMissingVarsRec(B, next_expected_var)
    if B = 0 or B = 1 then
        return n - next_expected_var + 1
    else
        B_node ← BDDPool[B]
        current_var ← Label(B_node)

        // Count skipped variables
        skipped ← current_var - next_expected_var
```

```
        low_missing ← NumMissingVarsRec(Low(B_node), current_var + 1)
        high_missing ← NumMissingVarsRec(High(B_node), current_var + 1)

        return skipped + max(low_missing, high_missing)
    end if
end function
```

Weekly Exercise 4.2

> Write the pseudo-code for the algorithm Until(T, Good, Target).

Solution:

```
function Until(T, Good, Target)
    // Initialize
    Reach ← Init
    New ← Init
    States ← [Init]  // Track states at each level for witness construction

    // Compute reachable states while staying in Good
    while not IsEmpty(New) do
        // Check if Target reached
        Reached ← Intersect(New, Target)
        if not IsEmpty(Reached) then
            // Construct witness path
            return ConstructWitness(States, Reached, Target)
        end if

        // Compute next states that stay in Good
        NextNew ← Intersect(Post(New, Trans), Good)
        NextNew ← Diff(NextNew, Reach)

        // Update sets
        if IsEmpty(NextNew) then
            // Cannot reach Target within Good
            return ∅
        end if

        Reach ← Union(Reach, NextNew)
        New ← NextNew
        States.append(New)
    end while
```

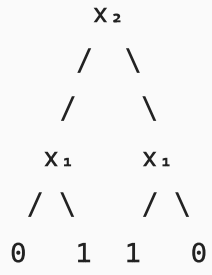
```
    return ∅  // Target not reachable within Good
end function

function ConstructWitness(States, Reached, Target)
    path ← []
    state ← PickState(Reached)
    path.prepend(state)

    // Work backwards from target state to initial state
    for i from States.length-1 downto 0 do
        // Find predecessor in previous level
        pred_set ← Intersect(Pre({state}, Trans), States[i])

        if not IsEmpty(pred_set) then
            pred ← PickState(pred_set)
            path.prepend(pred)
            state ← pred
        else
            // Should not reach here if algorithm is correct
            return ∅
        end if
    end for

    return path
end function
```

Binary Decision Diagrams

Exercise 1 (ex-1023.pdf)

> Draw a BDD for the formula XOR(x_2, x_1), where XOR is the exclusive-or operator.

Solution: The BDD for XOR(x_2, x_1) with variable ordering $x_2 < x_1$:

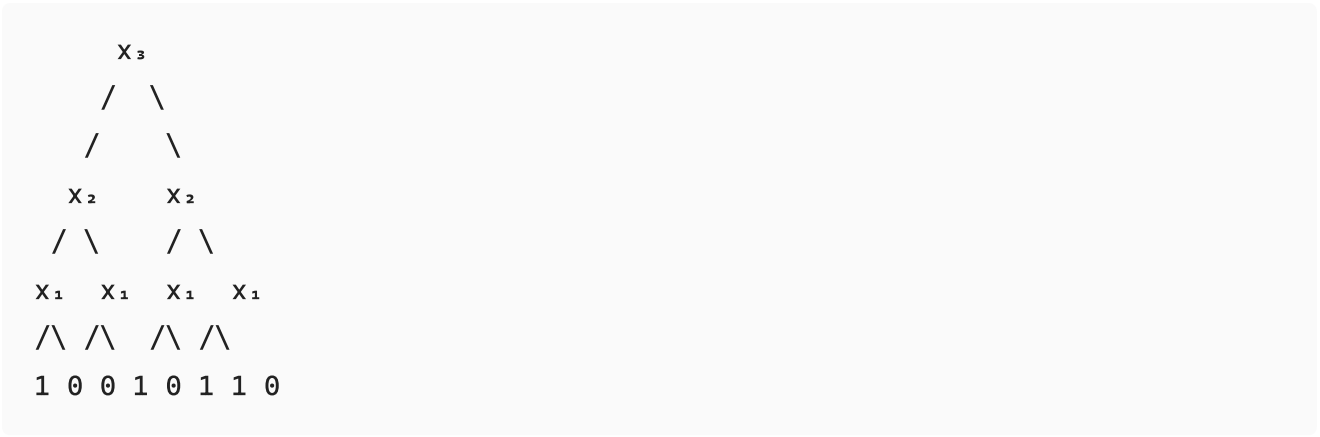


Exercise 2 (ex-1023.pdf)

> Draw a BDD for the formula XOR(x_3, XOR(x_2, x_1)).

Solution: The BDD for XOR(x_3, XOR(x_2, x_1)) with variable ordering $x_3 < x_2 < x_1$:

This can be reduced to:



Exercise 3 (ex-1023.pdf)

> How many x_1 nodes are in a (reduced) BDD for $x_4 \oplus x_3 \oplus x_2 \oplus x_1$?

Solution: In an n-variable parity function, each level i has 2^(i-1) nodes. For $x_4 \oplus x_3 \oplus x_2 \oplus x_1$ with ordering $x_4 < x_3 < x_2 < x_1$, level x_1 will have 2^(4-1) = 2^3 = 8 nodes.

Exercise 4 (ex-1023.pdf)

> How many total nodes are there in the (reduced) BDD for $x_n \oplus ... \oplus x_1$?

Solution: For an n-variable parity function:

- Level 1 (x_n): 1 node
- Level 2 (x_{n-1}): 2 nodes
- Level 3 (x_{n-2}): 4 nodes
- ...
- Level n (x_1): 2^(n-1) nodes

Total number of nodes = 1 + 2 + 4 + ... + 2^(n-1) = 2^n - 1

Exercise 1 (ex-1104.pdf)

> Write the pseudo-code for the function Exists(B, x) that takes as inputs a ROBDD B, a variable x, and returns a ROBDD representing the boolean function ∃x.f(B).

Solution:

```
function Exists(B, x)
    // Base cases
    if B = 0 or B = 1 then
        return B
    end if

    // Get node information
    B_node ← BDDPool[B]
    var ← Label(B_node)

    // If current variable comes before x in ordering
    if var < x then
        low ← Exists(Low(B_node), x)
        high ← Exists(High(B_node), x)
        return AddVertex(var, low, high)
    // If current variable is x
    else if var = x then
        // ∃x.f = f[x↦0] ∨ f[x↦1]
        return Or(Low(B_node), High(B_node))
    // If current variable comes after x in ordering
    else
        // x not in this path
        return B
    end if
end function
```

Exercise 2 (ex-1104.pdf)

> Write the pseudo-code for the function Restrict(B, x, b) that takes a ROBDD B, a variable x and a boolean value b ∈ {0, 1}, and returns the ROBDD representing the function f(B)[x ↦ b].

Solution:

```
function Restrict(B, x, b)
    // Base cases
    if B = 0 or B = 1 then
```

```
        return B
    end if

    // Get node information
    B_node ← BDDPool[B]
    var ← Label(B_node)

    // If current variable comes before x in ordering
    if var < x then
        low ← Restrict(Low(B_node), x, b)
        high ← Restrict(High(B_node), x, b)
        return AddVertex(var, low, high)
    // If current variable is x
    else if var = x then
        // Return appropriate child based on b
        if b = 0 then
            return Restrict(Low(B_node), x, b)
        else
            return Restrict(High(B_node), x, b)
        end if
    // If current variable comes after x in ordering
    else
        // x not in this path
        return B
    end if
end function
```

Exercise 3 (ex-1104.pdf)

> Write the pseudo-code for the function Rename(B, x_i, x_j) that returns a ROBDD that represents the function f(B)[$x_i \mapsto x_j$].

Solution:

```
function Rename(B, x_i, x_j)
    // Base cases
    if B = 0 or B = 1 then
        return B
    end if

    // Use Shannon expansion:
    // f[x_i ↦ x_j] = (¬x_j ∧ f[x_i↦0]) ∨ (x_j ∧ f[x_i↦1])
```

```
    // Compute f[x_i↦0] and f[x_i↦1]
    f_0 ← Restrict(B, x_i, 0)
    f_1 ← Restrict(B, x_i, 1)

    // Create ¬x_j and x_j BBDs
    not_x_j ← AddVertex(x_j, 1, 0)  // ¬x_j
    x_j ← AddVertex(x_j, 0, 1)     // x_j

    // Compute (¬x_j ∧ f[x_i↦0])
    term1 ← And(not_x_j, f_0)

    // Compute (x_j ∧ f[x_i↦1])
    term2 ← And(x_j, f_1)

    // Return (term1 ∨ term2)
    return Or(term1, term2)
end function
```

Temporal Logic and Büchi Automata

Exercise 1 (ex-1109.pdf)

> For each of the pair of formulas below, say whether the two are equivalent and if not whether one of them is stronger than the other.

Solution:

1. $\blacklozenge(\varphi_1 \wedge \varphi_2)$ and $(\blacklozenge\varphi_1 \wedge \blacklozenge\varphi_2)$
   - Not equivalent
   - $\blacklozenge(\varphi_1 \wedge \varphi_2)$ is stronger than $(\blacklozenge\varphi_1 \wedge \blacklozenge\varphi_2)$
   - Counterexample: Consider a trace where φ_1 is true only at position 1 and φ_2 is true only at position 2. Then $\blacklozenge\varphi_1 \wedge \blacklozenge\varphi_2$ is true, but $\blacklozenge(\varphi_1 \wedge \varphi_2)$ is false.
2. $\blacklozenge(\varphi_1 \vee \varphi_2)$ and $(\blacklozenge\varphi_1 \vee \blacklozenge\varphi_2)$
   - Equivalent
   - $\blacklozenge(\varphi_1 \vee \varphi_2)$ means "eventually either φ_1 or φ_2 is true"
   - $(\blacklozenge\varphi_1 \vee \blacklozenge\varphi_2)$ means "either eventually φ_1 is true or eventually φ_2 is true"
   - These are logically equivalent
3. $\square(\varphi_1 \wedge \varphi_2)$ and $(\square\varphi_1 \wedge \square\varphi_2)$
   - Equivalent
   - $\square(\varphi_1 \wedge \varphi_2)$ means "always both φ_1 and φ_2 are true"
   - $(\square\varphi_1 \wedge \square\varphi_2)$ means "always φ_1 is true and always φ_2 is true"
   - These are logically equivalent

4. $\Box(\varphi_1 \vee \varphi_2)$ and $(\Box\varphi_1 \vee \Box\varphi_2)$
    - Not equivalent
    - $(\Box\varphi_1 \vee \Box\varphi_2)$ is stronger than $\Box(\varphi_1 \vee \varphi_2)$
    - Counterexample: Consider a trace where φ_1 is true at even positions and φ_2 is true at odd positions. Then $\Box(\varphi_1 \vee \varphi_2)$ is true, but $(\Box\varphi_1 \vee \Box\varphi_2)$ is false.

Exercise 2 (ex-1109.pdf)

> We saw that the always-formula $\Box\varphi$ is equivalent to $\varphi \wedge \bigcirc\Box\varphi$. Find analogous formulas equivalent to the eventually-formula $\blacklozenge\varphi$ and to the until-formula $\varphi \, U \, \psi$.

Solution: For $\blacklozenge\varphi$:

- $\blacklozenge\varphi \equiv \varphi \vee \bigcirc\blacklozenge\varphi$
- Explanation: "Eventually φ" means either "φ is true now" or "φ will eventually be true starting from the next state"

For $\varphi \, U \, \psi$:

- $\varphi \, U \, \psi \equiv \psi \vee (\varphi \wedge \bigcirc(\varphi \, U \, \psi))$
- Explanation: "φ until ψ" means either "ψ is true now" or "φ is true now and φ until ψ holds from the next state"

Exercise 1 (ex-1113.pdf)

> Write an algorithm that takes as inputs a transition system T and a set of states F, and checks whether F is repeatable.

Solution:

```
function IsRepeatable(T, F)
    // Find all reachable states
    Reach ← Reachable(T)

    // Find reachable states in F
    ReachF ← Intersect(Reach, F)

    // For each state in ReachF, check if it's in a cycle
    foreach state s in ReachF do
        // Try to find a cycle containing s
        visited ← ∅
        stack ← [s]

        while not Empty(stack) do
            current ← Pop(stack)
```

```
            visited ← visited ∪ {current}

            t ← FirstSuccState(T, current)
            while t ≠ null do
                if t = s then
                    // Found a cycle containing s
                    return ConstructWitness(T, s)
                end if

                if t ∉ visited then
                    Push(stack, t)
                end if

                t ← NextSuccState(T, current, t)
            end while
        end while
    end foreach

    return false
end function

function ConstructWitness(T, s)
    // Find a path from an initial state to s
    init_path ← FindPathToState(T, s)

    // Find a cycle from s back to itself
    cycle_path ← FindCycle(T, s)

    return Concatenate(init_path, cycle_path)
end function
```

Exercise 1 (ex-1116.pdf)

> For each of the LTL-formulas below, construct a Büchi automaton that accepts exactly the traces that satisfy the formula.

Solution:

1. ♦e ∨ ♦f

```
q₀ (initial) → q₁ (accepting) : e
q₀ → q₂ (accepting) : f
q₀ → q₀ : ¬e ∧ ¬f
```

```
q₁ → q₁ : true
q₂ → q₂ : true
```

2. ♦e ∧ ♦f

```
q₀ (initial) → q₁ : e
q₀ → q₀ : ¬e
q₁ → q₂ (accepting) : f
q₁ → q₁ : ¬f
q₂ → q₂ : true
```

3. □(e → e U f)

```
q₀ (initial, accepting) → q₀ : ¬e ∨ f
q₀ → q₁ : e ∧ ¬f
q₁ → q₁ : e ∧ ¬f
q₁ → q₀ : f
```

Exercise 2 (ex-1116.pdf)

> Consider the following nondeterministic Büchi automaton for the LTL formula Persistenly e. Can we construct an equivalent deterministic Büchi automaton?

Solution: The given automaton accepts traces where eventually e holds forever (□♦e).

No, we cannot construct an equivalent deterministic Büchi automaton. This is because the language "eventually always e" (□♦e) is not expressible as a deterministic Büchi automaton.

The key insight is that deterministic Büchi automata can express safety properties and some liveness properties, but not all liveness properties. Specifically, they cannot express fair properties like "infinitely often e" (□♦e).

Exercise 1 (ex-1118-1.pdf)

> Write an algorithm that takes as inputs a symbolic transition system T = (S, Init, Trans) and a region A, and returns the region {s ∈ A | there exists t ∈ A that is reachable from s in ≥ 1 transitions}.

Solution:

```
function ComputeCanReach(T, A)
    Result ← ∅
    New ← A
```

```
    while not IsEmpty(New) do
        Next ← Intersect(Post(New, Trans), A)
        if IsEmpty(Next) then
            break
        end if

        Result ← Union(Result, New)
        New ← Diff(Next, Result)
    end while

    return Result
end function
```

Exercise 2 (ex-1118-1.pdf)

> Use the algorithm of Exercise 1 to write a symbolic algorithm that takes as inputs a symbolic transition system T = (S, Init, Trans) and a region F. If F is repeatable in T your algorithm should return True. Otherwise, your algorithm should return False.

Solution:

```
function IsRepeatable(T, F)
    // 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
        // Find states in Recur that can reach some state in Recur in ≥1
steps
        CanReach ← ComputeCanReach(T, Recur)

        if not IsEmpty(CanReach) then
            return True
        end if

        // Update Recur to states that are in a potential cycle
```

```
        Recur ← Intersect(Recur, CanReach)
    end while

    return False
end function
```

Exercise 1 (ex-1118-2.pdf)

> The symbolic algorithm for repeatability uses both post-image computation and pre-image computation. Suppose we modify the algorithm by replacing both calls to Pre by Post... How will this modification impact the correctness of the algorithm?

Solution: The modified algorithm would be incorrect. Using Post instead of Pre changes the meaning from "states that can reach Recur" to "states that can be reached from Recur."

For the given example:

- Original algorithm correctly identifies that F is not repeatable (there are no cycles)
- Modified algorithm would incorrectly conclude that F is repeatable because the PreReach set would include all states, and Recur would include all states

The issue is that the original algorithm checks if states in Recur can reach other states in Recur (forming a cycle), while the modified algorithm just checks if states in Recur can reach any state.

Exercise 2 (ex-1118-2.pdf)

> Prove that the symbolic algorithm is correct.

Solution: To prove correctness, we need to show two directions:

1. If F is repeatable, the algorithm returns True:
   - If F is repeatable, there exists a reachable state $s \in F$ such that s is in a cycle
   - This state s will be in the initial Recur set
   - Since s is in a cycle, there's a path of ≥1 transitions from s back to itself
   - This means s is in PreReach, and so s remains in Recur in each iteration
   - Eventually, the algorithm will determine that Recur is a subset of PreReach, returning True
2. If the algorithm returns True, F is repeatable:
   - The algorithm returns True only when Recur becomes a subset of PreReach
   - This means that for every state s in Recur, there is a path from s to some state t in Recur
   - By iterating this property, we can construct an infinite path that visits Recur infinitely often

- Since Recur ⊆ F, this path visits F infinitely often, making F repeatable

Exercise 1 (ex-1123.pdf)

> Consider the LTL formula φ = (e U ○f) ∨ ¬e. First compute the closure Sub(φ). Then apply the tableau construction to build the generalized Büchi automaton Mφ.

Solution: First, compute Sub(φ):

- φ = (e U ○f) ∨ ¬e
- Sub(φ) includes:
  - φ, ¬φ
  - e U ○f, ¬(e U ○f)
  - ¬e, e
  - ○f, ¬○f
  - f, ¬f
  - ○(e U ○f), ¬○(e U ○f)

The tableau construction yields a generalized Büchi automaton with states representing maximal consistent subsets of Sub(φ). The reachable states would include:

- State 1: {φ, ¬e, ¬(e U ○f), ...}
- State 2: {φ, e, e U ○f, ○f, ...}
- State 3: {φ, e, e U ○f, ¬○f, ○(e U ○f), ...}

With transitions based on next-state formulas and accepting conditions for each until formula.

Exercise 2 (ex-1123.pdf)

> Given a Generalized Büchi automaton M over a set V of input variables, show how to build a (standard) Büchi automaton M' over the same input variables V that accepts the same language of M.

Solution: To convert a generalized Büchi automaton $M = (V, Q, Init, \{F_1, F_2, ..., F_k\}, E)$ into a standard Büchi automaton M':

1. Create k+1 copies of the state space: $Q \times \{0, 1, ..., k\}$
2. For each state (q, i) and each edge $q \rightarrow q'$ in M:
   - If $q' \in F_i$, add an edge $(q, i) \rightarrow (q', (i+1) \bmod (k+1))$
   - Otherwise, add an edge $(q, i) \rightarrow (q', i)$
3. Set initial states to $\{(q, 0) \mid q \in Init\}$
4. Set accepting states to $\{(q, 0) \mid q \in Q\}$

This construction ensures that an infinite run is accepting in M' if and only if it visits all accepting sets F_1, F_2, ..., F_k in M infinitely often.

Controller Synthesis

Exercise 1 (ex-1213.pdf)

> Two philosophers are seated at a round table where there are two plates of food, one in front of each philosopher, and two forks. Provide a model for the two philosophers and the forks.

Solution: Philosopher 1 automaton:

```
States: {Thinking, HasFork1, HasFork2, Eating}
Initial state: Thinking
Transitions:
  Thinking → HasFork1: p1f1
  Thinking → HasFork2: p1f2
  HasFork1 → Eating: p1f2
  HasFork2 → Eating: p1f1
  Eating → Thinking: r1
```

Philosopher 2 automaton:

```
States: {Thinking, HasFork1, HasFork2, Eating}
Initial state: Thinking
Transitions:
  Thinking → HasFork1: p2f1
  Thinking → HasFork2: p2f2
  HasFork1 → Eating: p2f2
  HasFork2 → Eating: p2f1
  Eating → Thinking: r2
```

Fork 1 automaton:

```
States: {Free, UsedBy1, UsedBy2}
Initial state: Free
Transitions:
  Free → UsedBy1: p1f1
  Free → UsedBy2: p2f1
  UsedBy1 → Free: r1
  UsedBy2 → Free: r2
```

Fork 2 automaton:

```
States: {Free, UsedBy1, UsedBy2}
Initial state: Free
Transitions:
  Free → UsedBy1: p1f2
  Free → UsedBy2: p2f2
  UsedBy1 → Free: r1
  UsedBy2 → Free: r2
```

Exercise 2 (ex-1213.pdf)

> Consider your solution to Exercise 1 about the two philosophers. Establish which states of the parallel composition of the four automata are reachable, which states have deadlock, and which states are blocking.

Solution: The reachable states of the parallel composition include:

- (Thinking, Thinking, Free, Free)
- (HasFork1, Thinking, UsedBy1, Free)
- (HasFork2, Thinking, Free, UsedBy1)
- (Thinking, HasFork1, UsedBy2, Free)
- (Thinking, HasFork2, Free, UsedBy2)
- (Eating, Thinking, UsedBy1, UsedBy1)
- (Thinking, Eating, UsedBy2, UsedBy2)
- (HasFork1, HasFork2, UsedBy1, UsedBy2)
- (HasFork2, HasFork1, UsedBy2, UsedBy1)

Deadlock states:

- (HasFork1, HasFork2, UsedBy1, UsedBy2): P1 has fork 1 and P2 has fork 2, neither can proceed
- (HasFork2, HasFork1, UsedBy2, UsedBy1): P1 has fork 2 and P2 has fork 1, neither can proceed

Blocking states: The deadlock states are also blocking states, as they cannot reach a marked state.

Exercise 3 (ex-1213.pdf)

> Add an automaton to the model of the dining philosophers that prevents deadlocks and blocking states from occurring.

Solution: We can add a supervisor automaton:

```
States: {Both, OnlyP1, OnlyP2}
Initial state: Both
Transitions:
  Both → OnlyP1: p1f1, p1f2
  Both → OnlyP2: p2f1, p2f2
  OnlyP1 → Both: r1
  OnlyP2 → Both: r2
```

This supervisor ensures that only one philosopher can pick up forks at a time, preventing the deadlock situation.

Exercise 4 (ex-1213.pdf)

> Compare your solution with the solutions of the other students. Do they allow the same set of behaviours?

Solution: Different solutions might include:

1. Resource hierarchy (numbering forks)
2. Dining philosophers with conductor
3. Chandy/Misra solution

These solutions might differ in permissiveness:

- The resource hierarchy solution allows more concurrency than strict mutual exclusion
- The conductor solution is less permissive but simpler
- The solution with a supervisor preventing both philosophers from having forks simultaneously is the least permissive

Exercise 5 (ex-1213.pdf)

> Use the basic supervisory control synthesis algorithm to compute the maximally permissive proper supervisor for the dining philosophers system.

Solution: Using the basic supervisory control synthesis algorithm:

1. Start with the uncontrolled plant (philosophers and forks)
2. Compute the set of blocking states
3. Compute the set of bad states (blocking states and states that can reach a bad state via uncontrollable events)
4. Remove transitions with controllable events that target bad states
5. Remove unreachable states and transitions

The resulting supervisor prevents both philosophers from having exactly one fork each, as this leads to a deadlock. The supervisor disables the transitions that would lead to these

states.

Exercise 1 (ex-1215.pdf)

> Use the basic supervisory control synthesis algorithm to compute the maximally permissive proper supervisor for the dining philosophers system.

Solution: This is the same as Exercise 5 from ex-1213.pdf.

The maximally permissive proper supervisor allows transitions where:

- If both forks are free, either philosopher can pick up either fork
- If philosopher 1 has fork 1, philosopher 2 cannot pick up fork 2
- If philosopher 1 has fork 2, philosopher 2 cannot pick up fork 1
- If philosopher 2 has fork 1, philosopher 1 cannot pick up fork 2
- If philosopher 2 has fork 2, philosopher 1 cannot pick up fork 1

Exercise 2 (ex-1215.pdf)

> The solution to the dining philosophers problem you obtained on Exercise 1 suffers from individual starvation. Define as a requirement that every philosopher regularly gets a possibility to eat if he wants to.

Solution: The requirement that every philosopher regularly gets a possibility to eat is a liveness or progress property, not a safety property.

We can define it as:

- For philosopher 1: $\Box(\text{Thinking}_1 \rightarrow \blacklozenge\text{Eating}_1)$
- For philosopher 2: $\Box(\text{Thinking}_2 \rightarrow \blacklozenge\text{Eating}_2)$

To enforce this property, we need to modify the supervisor to ensure fairness. One approach is to introduce a turn-based mechanism:

```
States: {Turn1, Turn2}
Initial state: Turn1
Transitions:
  Turn1 → Turn2: r1
  Turn2 → Turn1: r2
```

The supervisor would only allow a philosopher to pick up forks when it's their turn, ensuring that starvation cannot occur.