CS4211: Formal Methods for Software Engineering

Lecture Notes

Kevin Toh

Acknowledgement

These set of slides are modified from the content and lecture notes of the following professors:

- Professor Dong Jing Song
- Professor Yang Liu
- Professor Jun Sun
- Professor Abhik Roychoudhury
- Professor Jiang Kan

I would like to express my gratitude for their invaluable lecture notes and content that forms the basis of this set of lecture notes.

Table of Contents

Introduction to Formal Methods

- Requirements are difficult to define because its written in **Natural Language** which can be imprecise and ambiguous at times.
- As we cannot anticipate the ways a system may be used, written test cases only covers a small subset of use cases.
- We want to verify if a system always satisify a certain property, in possible all cases.
- In Formal Methods, we use **Mathematics** to define the **structure** and **behaviour** of our software because it is **precise** and **unambiguous**
- Eventually, we can use a model checker to automatically verify the software by checking if a certain property holds in all cases.

Latex Setup

- The Latex package that we will be using is zed-csp
- **Setup Code:**

```
% For latex document
\documentstyle[12pt,zed]{article}

% For beamer slides
\usepackage{zed-csp}

\begin{document}
\end{document}
```

- Reference: https://sg.mirrors.cicku.me/ctan/macros/latex/contrib/zed-csp/zed2e.pdf

Table of Contents

The Z Specification Language

- Based on set theory and mathematical logic
  - We will be doing a recap on predicates, set theory, functions and relations next.
- Uses schemas to declare object properties
  - Schemas are similar to defining the structure of a class and its properties in an Object-Oriented Programming Language.
- Uses operations to describe state transitions
  - Each object has a state representing the values it's properties hold at a certain moment in time.
  - Operations are similar to methods of a class.
  - Operations modify the state of an object.
  - We use predicates to describe state transitions in an operation.
- We can then proof that a certain property holds manually

Recap on Predicates and Logic

Predicate

A statement that is either true or false.

1. There are 365 days in 2024. (**false**)
2. Let $P(x, y)$ be $x + y = 9$
   - $P(4, 5)$ is **true**.
   - $P(3, 7)$ is **false**.

Logic Operators

1. Not ($\neg$) Eg:
2. And ($\wedge$)
3. Or ($\vee$)
4. Implies ($\Rightarrow$)
5. Equivalence ($\Leftrightarrow$)

Recap on Quantifiers

1. Universal Quantifier ($\forall$)
   - Example: All natural numbers are greater than -1.
   - Mathematically, we would write $\forall n \in \mathbb{N}, n > -1$
   - In Z Specification, we would write $\forall n : \mathbb{N} \bullet n > -1$
   - $\forall n : \mathbb{N} \bullet n > 0$
   - In general, $\exists x : X \bullet P(x)$ abbreviates $P(a) \land P(b) \land P(c) \land \ldots$
2. Existential Quantifier ($\exists$)
   - Example: There exists a natural number more than 0.
   - In Z Specification, we would write $\exists n : \mathbb{N} \bullet n > 0$
   - In general, $\exists x : X \bullet P(x)$ abbreviates $P(a) \lor P(b) \lor P(c) \lor \ldots$

Differences between Mathematical Notations and Z Specification

- In Mathemical Notation, : or | means "such that" when used in Set Expressions.
- In Z Specification, : means "belongs to".
  - The difference between : and $\in$ will be explained later.
- In Z Specification, $\bullet$ means "such as" when writing predicates.

Recap on Set Theory

- A set is a collection of elements (or members)
  - Elements are not ordered: $\{a, b, c\} = \{b, a, c\}$
  - Elements are not repeated" $\{a, a, b\} = \{a, b\}$
  - Given Sets
    - $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ (The set of all natural numbers)
    - $\mathbb{N}_1 = \{1, 2, 3, \ldots\}$
    - $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ (The set of all integers)
    - $\mathbb{R}$ (The set of all real numbers)
    - $\emptyset$ (Empty Set: The set with no elements)
- Membership: $x \in \mathbb{X}$ is a predicate which is
  - true if x is in the set $\mathbb{X}$. Eg: $a \in \{a, b, c\}$
  - false if x is not in the set $\mathbb{X}$. Eg: $d \in \{a, b, c\}$

Difference between ':' and '$\in$'

Example: $\forall x : \mathbb{Z} \bullet x > 5 \Rightarrow x \in \mathbb{N}$

- $x : \mathbb{Z}$ declares a new variable x of type $\mathbb{Z}$
- $x \in \mathbb{N}$ is a predicate which is either true or false depending on the value of the declared x.

Recap on Set Theory

- Set Expressions
  - We can express a set by listing its elements if the set is finite and small.
    - $\{a, b, c, d\}$ is a finite set.
  - If a set is large or infinite, we can definite a set by giving a predicate which specifies precisely those elements in a set.
    - $\mathbb{N}$ is an infinite set.
    - The set of natural numbers less than 99 is $\{n : \mathbb{N} \mid n < 99\}$
    - In general the set $\{x : \mathbb{X} \mid P(x)\}$ is the set of elements of $\mathbb{X}$ for which predicate P is true.
- Set Examples
  - The set of even integers is $\{z : \mathbb{Z} \mid \exists\, k : \mathbb{Z} \bullet z = 2k\}$
  - The set of natural numbers which when divided by 7 leave a remaineder of 4 is $\{n : \mathbb{N} \mid \exists\, m : \mathbb{N} \bullet n = 7m + 4\}$
  - $\mathbb{N}$ is the set $\{z : \mathbb{Z} \mid z \geq 0\}$
  - $\mathbb{N}_1$ is the set $\{n : \mathbb{N} \mid n \geq 1\}$
  - If a, b are any natural numbers, then $a \,..\, b$ is defined as the set of all natural numbers between a and b inclusive.
    - $a \,..\, b$ is the set $\{n : \mathbb{N} \mid a \leq n \leq b\}$

Recap on Set Theory

- Subset ($\subseteq$): If S and T are sets, $S \subset T$ is a predicate equivalent to $\forall s : S \bullet s \in T$.
  - The following predicates are true:
    - $\{0, 1, 2\} \subseteq \mathbb{N}$
    - $2 \mathinner{..} 3 \subseteq 1 \mathinner{..} 5$
    - $\{a, b\} \subseteq \{a, b, c\}$
    - $\emptyset \subseteq X$ for any set X
    - $\{x\} \subseteq X \Leftrightarrow x \in X$
- Proper Subset ($\subset$): If S and T are sets, $S \subset T$ is a predicate equivalent to $S \subseteq T \land S \neq T$.
- Power Set ($\mathbb{P}$): If X is a set, $\mathbb{P}\ X$ (the power set of X) is the set of all subsets of X.
  - $A \in \mathbb{P}\ B = A \subseteq B$
  - The following predicates are true:
    - $\mathbb{P}\{a, b\} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$
    - $\mathbb{P}\ \emptyset = \{\emptyset\} \neq \emptyset$
    - $1..5 \in \mathbb{P}\ \mathbb{N}$
    - $2..5 \in \mathbb{P}(1..5)$
  - If X has k elements, then $\mathbb{P}\ X$ has 2^k elements.

Recap on Set Theory

- Set Operations
  - Set Union: Suppose $S, T : \mathbb{P}\, X$ or $S \subseteq X, T \subseteq X$, then $S \cup T = \{x : X \mid x \in S \vee x \in T\}$
    - $\{a, b, c\} \cup \{b, g, h\} = \{a, b, c, g, h\}$
    - $A \cup \emptyset = A$ (for any set A)
  - Set Intersection: Suppose $S, T : \mathbb{P}\, X$, then $S \cap T = \{x : X \mid x \in S \wedge x \in T\}$
    - $\{a, b\} \cap \{b, c\} = \{b\}$
    - $\{a, b, c\} \cap \{d, g\} = \emptyset$ (disjoint sets)
    - $A \cap \emptyset = \emptyset$ (for any set A)
  - Set Difference: Suppose $S, T : \mathbb{P}\, X$, then $S - T = \{x : X \mid x \in S \wedge x \notin T\}$
    - $\{a, b, c\} - \{b, g, h\} = \{a, c\}$
    - $\mathbb{N}_1 = \mathbb{N} = \{0\}$
  - Cartesian Product: If A and B are sets, then $A \times B$ is the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$.
    - $\{a, b\} \times \{a, c\} = \{(a, a), (a, c), (b, a), (b, c)\}$
  - Cardinality: $\#X$ is a natural number denoting the cardinality of (number of elements in) a finite set X.
    - $\#\{a, b, c\} = 3$

Tuples and Cartesian Product

- An n-tuple $(x_1, \ldots, x_n)$ is present in the Cartesian Product $\mathbf{a_1} \times \ldots \times \mathbf{a_n}$ if and only if each element x_i is an element of the corresponding set $\mathbf{a_i}$.
- To refer to a particular component of a tuple t, we use the projection notation $(.)$
- Suppose we have $t = (x_1, x_2, \ldots, x_n)$
  - The first component of the tuple t is written as $t.1$ which is the value $\mathbf{x_1}$.
  - The second component of the tuple t is written as $t.2$ which is the value $\mathbf{x_2}$.
  - The n-th component of the tuple t is written as $t.n$ which is the value $\mathbf{x_n}$

Recap on Relations

- A relation R from sets A to B, is declared as $R : A \leftrightarrow B$ is a subset of $A \times B$
- Example: $R = \{(c, x), (c, z), (d, x), (d, y), (d, z)\}$
  - The following predicates are equivalent
    1. $(c, z) \in R$
    2. $c \rightarrow z \in R$
    3. cRz
- **Domain:** $\mathrm{dom}\, R$ is the set $\{a : A \mid \exists\, b : B \bullet aRb\}$
- **Range:** $\mathrm{ran}\, R$ is the set $\{b : B \mid \exists\, a : A \bullet aRb\}$

Types in Z Specification

- Z specification language is **strongly typed**.
- Every expression is given a type.
- Any set can be used as a type.
- The following are equivalent declarations of variables x and y of types A and B respectively.
  - $(x, y) : A \times B$
  - $x : A, y : B$
  - $x, y : A$ (only when $B = A$)

Modelling using Z Specification

- When we write a program, we can write code procedurally, functionally or in an object oriented manner.
- Z Specification can help us model our code using two distinct sections.
  1. Declaration: To define variables.
  2. Predicate: Often used to define behaviours or invariants.
- **Example #1:** We can define the relation **divides** between two natural numbers.

  | `divides:` $\mathbb{N}_1 \leftrightarrow \mathbb{N}$

  | ----------

  | $\forall x : \mathbb{N}_1;\ y : \mathbb{N} \bullet x \ \underline{\texttt{divides}}\ y \Leftrightarrow \exists k : \mathbb{N} \bullet x \cdot k = y$

  Usage: 3 divides 6, $\neg$ (3 divides 7)
- **Example #2:** We can define the relation $\leq$ between two natural numbers.

  | $\_ <= \_ :$ $\mathbb{N} \leftrightarrow \mathbb{N}$

  | ----------

  | $\forall x, y : \mathbb{N} \bullet x <= y \Leftrightarrow \exists k : \mathbb{N} \bullet x + k = y$

  The relation $<=$ is the infinite subset of ordered pairs in $\mathbb{N} \times \mathbb{N}$.

  $\{(0,0), (0,1), (1,1), (0,2)(1,2), (2,2), \ldots\}$

Domain and Range Restriction

- Let A, B, R, S, T be sets.
- A is the domain set, B is the range set and R is the relation set.
- Note that S is a subset of the domain set and T is the subset of the range set.
- Suppose $R : A \leftrightarrow B, S \subseteq A$ and $T \subseteq B$.
  - **Domain Restriction:** $S \lhd R$ is the set $\{(a, b) : R \mid a \in S\}$
  - **Range Restriction:** $R \rhd T$ is the set $\{(a, b) : R \mid b \in T\}$
- Notice that both $S \lhd R \in A \leftrightarrow B$ and $R \rhd T \in A \leftrightarrow B$, meaning that both domain restriction and range restrictions are relations from sets A to B.
- If has_sibling: People $\leftrightarrow$ People then
  - female $\lhd$ has_sibling is the relationship is_sister_of.
  - has_sibling $\rhd$ female is the relationship has_sister.

Domain and Range Subtraction

- Let A, B, R, S, T be sets.
- A is the domain set, B is the range set and R is the relation set.
- Note that S is a subset of the domain set and T is the subset of the range set.
- Suppose $R : A \leftrightarrow B, S \subseteq A$ and $T \subseteq B$.
  - **Domain Subtraction:** $S \lhd R$ is the set $\{(a, b) : R \mid a \notin S\}$
  - **Range Subtraction:** $R \rhd T$ is the set $\{(a, b) : R \mid b \notin T\}$
- The following predicates are true.
  - $S \lhd R = (A - S) \lhd R$
  - $R \rhd T = R \rhd (B - T)$
  - $S \lhd R \in A \leftrightarrow B$
  - $R \rhd T \in A \leftrightarrow B$
- If has_sibling: People $\leftrightarrow$ People then
  - female $\lhd$ has_sibling is the relationship is_brother_of.
  - has_sibling $\rhd$ female is the relationship has_brother.

Relational Image

- Suppose the relation $R : A \leftrightarrow B$ and $S \leftrightarrow A$
- $R(\!| S |\!) = \{b : B \mid \exists a : S \bullet aRb\}$
- $R(\!| S |\!) \leftrightarrow B$
- Example
  - $divides(\!| \{8, 9\} |\!) = \{x : \mathbb{N} \mid \exists k : \mathbb{N} \bullet x = 8 \cdot k \lor 9 \cdot k\} = \{0, 8, 9, 16, 18, \ldots\}$
  - $<= (\!| 3, 7, 21 |\!) = \{x : \mathbb{N} \mid x >= 3\}$
- In summary, the relational image returns the set of all elements $b \in B$ such that there exists an $a \in S$ with $(a, b) \in R$.
- The difference between relational image and range restriction is that range restriction returns the subset of R which are ordered pairs of (a, b) where $a \in A$ and $b \in B$ and the first element a of the ordered pair is in S. The relational image simply just returns the set of all second elements b.

Inverse and Relational Composition

- **Inverse:** R^{-1} is the set $\{(b, a) : B \times A \mid aRb\}$ or $R^{-1} \in B \leftrightarrow A$
  - $has\_sibling^{-1} = has\_sibling$
  - $divisor^{-1} = has\_divisor$
- Example: $succ^{-1} = \text{pred}$

  | succ : $\mathbb{N} \leftrightarrow \mathbb{N}$

  |----------

  | $\forall x, y : \mathbb{N} \bullet x \ \underline{\text{succ}} \ y \leftrightarrow x + 1 = y$
- **Relational Composition ($\fatsemi$)**
  - Suppose $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$ are two relations.
  - $R \fatsemi S = \{(a, c) : A \times C \mid \exists\, b : B \mid aRb \land bSc\}$
  - $R \fatsemi S \in A \leftrightarrow C$
- Examples
  - $is\_parent\_of \fatsemi is\_parent\_of = is\_grandparent\_of$
  - $R^0 = id[A]$
  - $R^1 = R$
  - $R^2 = R \fatsemi R$
  - $R^3 = R \fatsemi R \fatsemi R$

Recap on Functions

- A (partial) function from a set A to a set B, denoted by $f : A \pfun B$ is a subset f of $A \times B$ with the property that for each $a \in A$, there is **at most one** $b \in B$ with $(a, b) \in f$.
- $\dom f$ is the set $\{a : A \mid \exists\, b : B \bullet (a, b) \in f\}$
- $\ran f$ is the set $\{b : B \mid \exists\, a : A \bullet (a, b) \in f\}$
- Suppose $f : A \pfun B$ and $a \in \dom f$, then $f(a)$ denotes the unique image $b \in B$ that a is mapped to by f.
- $(a, b) \in f$ is equivalent to $f(a) = b$
- **Total Function:** If the function $f : A \pfun B$ is a total function, then $f : A \fun B$ if and only if $\dom f = A$

Function Overriding

- Suppose $f, g : A \pfun B$, then $f \oplus g$ is the function $(\dom g \domsub f) \cup g$.
- The following predicates are true:
  1. $\dom f \oplus g = \dom f \cup \dom g$
  2. $a : \dom g \bullet (f \oplus g)(a) = g(a)$
  3. $\forall a : \dom f - \dom g \bullet (f \oplus g)(a) = f(a)$
  4. $\mathrm{f} \oplus \mathrm{g} \in a \pfun b$
- Examples
  1. $\{a \mapsto x, b \mapsto y, c \mapsto x\} \oplus \{a \mapsto y\} = \{a \mapsto y, b \mapsto y, c \mapsto x\}$
  2. double $\oplus$ root $= \{(0,0), (1,1), (2,4), (3,6), (4,2), \ldots\}$ (Note: $(4,8)$ was replaced with $(4,2)$ as the domain 4 is both in f and g, so the range was replaced with 2 that was in g)

Specifying Functions

1. Using a look-up table
   - If a function $f : A \pfun B$ is finite (and not too large), we can specify the function explicity by listing all pairs (a, b) in the subset $A \times B$.
   - Example: PassportNo $\to$ Address

PassportNo	Address
A001017	77 Sunset Strip
...	...
G707165	19 Mail Street

2. Declaring Axioms: A function can be specified by giving a **predicate** determining which pairs (a, b) are in the function.
   - **Example: The root function that calculates the square root of a natural number**

     | `root` $\mathbb{N} \pfun \mathbb{N}$
     |---
     | $\mathrm{dom}\, root = \{n : \mathbb{N} \mid \exists m : \mathbb{N} \bullet m^2 = n\}$
     | $\forall n : \mathrm{dom}\, root \bullet (root(n))^2 = n$

Specifying Functions

3. Using Recursion: For functions defined recursively in terms of itself.

   | fact $\mathbb{N}_1 \pfun \mathbb{N}$
   |---
   | fact(1) = 1
   | $\forall n : \mathbb{N}_1 - \{1\} \bullet fact(n) = n * fact(n-1)$

4. Giving an Algorithm: A function $f : A \pfun B$ is specified by an algorithm such that given any element a in the domain of f, the element $f(a)$ can be computed using the algorithm.

```
1  input n : N
2  var x, y: integer;
3  begin
4      x := n; y:= 0;
5      while x != 0 do
6      begin
7          x := x - 1; y:= y + 2
8      end;
9  write(y)
10 end
```

| double: $\mathbb{N} \to \mathbb{N}$
| ---
| $\forall n : \mathbb{N} \bullet double(n) = 2n$

Sequences

- A sequence s of elements of a set A, denoted $s : \mathrm{seq}\, A$, is a function $s : \mathbb{N} \pfun A$ where $\mathrm{dom}\, s = 1 \mathinner{..} n$ for some natural number n.
- **Example**
  - $< b, c, a, b >$ denotes the sequence (function) $\{1 \mapsto b, 2 \mapsto c, 3 \mapsto a, 4 \mapsto b\}$
  - The empty sequence is denoted by $<>$
- The set of all sequences of elements from A is denoted as $\mathrm{seq}\, A$ and is defined to be $\mathrm{seq}\, A = \{s : \mathbb{N} \pfun A \mid \exists n : \mathbb{N} \bullet \mathrm{dom}\, s = 1 \mathinner{..} n\}$
- $\mathrm{seq}_1\, A = \mathrm{seq}\, A - \{<>\}$ is defined as the set of non-empty sequences.
- Since sequences are ordered mapping, $< a, b, a > \neq < a, a, b > \neq < a, b >$

Special Functions for Sequence

1. Concatenation
   - $< a, b > \frown < b, a, c > = < a, b, b, a, c >$
2. Head
   - $\mid$ `head:` $\quad \mathrm{seq}_1 A \to A$
     $\mid$`----------`
     $\mid \forall s : \mathrm{seq}_1 A \bullet head(s) = s(1)$
   - $head < c, b, b > = c$
3. Tail
   - $\mid$ `tail:` $\quad \mathrm{seq}_1 A \to \mathrm{seq}\, A$
     $\mid$`----------`
     $\mid \forall s : \mathrm{seq}_1 A \bullet < head(s) > \frown tail(s) = s$
   - $tail < c, b, b > = < b, b >$
4. Filter
   - $< a, b, c, d, e, d, c, b, a > \upharpoonright \{a, d\} = < a, d, d, a >$
   - Filter only keeps the element in the specified set, preserves order in the original sequence and outputs a new sequence.

Z Specification

- As explained in an earlier slide, we can write Z Specification to formally specify requirements in **two distinct sections**
  1. Declaration: To define variables.
  2. Predicate: Often used to restrain the possible values of the declared variables to define behaviours or invariants.
- Formal Specification can be done in the form of both **axioms** and **schemas**.
- When we were specifying functions earlier, we were formally specifying them in the form of **axioms** in the **axiom environment**.
- Later we will introduce how to formally specify **schemas** which are used to specify relationships between variable values.
- There are two main type of schemas in the **schema environment**
  - State Schema
  - Operation Schema

The zed-csp Package

- We can use the zed-csp package to formally specify requirements in Z Specification and render axioms and schemas them in TeX.
- There are **two types of environment** in the zed-csp package.

1. Axiom Environment

$$limit : \mathbb{N}$$
$$\hline$$
$$limit \leq 65535$$

```
\begin{axdef}
    limit: \nat
    \where
    limit \leq 65535
\end{axdef}
```

2. Schema Environment

PhoneDB

$$known : \mathbb{P}\, NAME$$
$$phone : NAME \pfun PHONE$$
$$\hline$$
$$known = \mathrm{dom}\, phone$$

```
\begin{schema}{PhoneDB}
known: \power NAME \\ phone: NAME \pfun PHONE
\where
known = \dom phone
\end{schema}
```

State Schema

- A **state schema** specifies a relationship between variable values
- It specifies a snapshot of a system
- **Variables** are declared and typed in the **top part of the schema**.
- A **predicate (axiom)** restraining the possible values of the declared variables are given in the **bottom part of the schema.**
- An instance of a schema is an assignment of values to variables consistent with their type declaration and satisfying the predicate.

Operation Schema

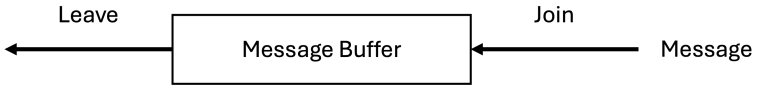
- The state schema provides a static view of the system.
- To specify how the system can change, we need to specify the operation schema.
- An operation can be thought as taking an instance of the state schema and producing a new instance.
- To specify such an operation, we express as a predicate the relationship between the **instance of the state before the operation** and the **instance after the operation.**

Convention

- The value of the state variables before the operation are denoted by **unprimed identifiers**.
  - Example: `items : seq MSG`
- Values after the operation are denoted by **primed identifiers**.
  - Example: `items' : seq MSG`
- Note: Let `MSG` be the set of all possible messages that can be transmitted.

Case Study: A Message Buffer

- We are going to model a message buffer to learn about **schema specification**.
- A message buffer stores messages within a queue data structure and operates on a first in / first out (FIFO) principle.
- Suppose we have a line which might be occupied by traffic, many messages join the buffer and but only the first message in the queue leaves the buffer when the line is free.
- The buffer may contain several messages at any time, but there is a fixed upper limit on the number of messages the buffer may contain.
- To model a message buffer, we minimally need the following:
  1. States: Buffer (to store messages)
  2. Operations: Join (new message joins the buffer), Leave (message leaves the buffer)



Formal Specification: Message Buffer

State Schema: Buffer

- Let MSG be the set of all possible messages that can be transmitted.
- Let max : $\mathbb{N}$ be the constant maximum number of messages that can be held in the buffer at any one time.
- Example: Let $MSG = \{m1, m2, m3\}$ and $max = 4$
  1. $items =< m1, m2 >$ is a valid instance.
  2. $items =< m3, m1, m1, m2, m2 >$ is an invalid instance.

Operation Schema: Join

- The decoration ? denotes an input
- There is an implicit $\wedge$ between each line in the predicate section.

Buffer

$items : \operatorname{seq} MSG$

$\#items \leq max$

Join

$items, items' : \operatorname{seq} MSG$
$msg? : MSG$

$\#items \leq max$
$\#items' \leq max$
$\#items < max$
$items' = items^\frown < msg? >$

Explanation: Predicate of the Join Operation

$$
\begin{array}{|l}
\textit{Join} \\
\hline
\textit{items}, \textit{items}' : \operatorname{seq} \textit{MSG} \\
\textit{msg}? : \textit{MSG} \\
\hline
\#\textit{items} \leq \textit{max} \\
\#\textit{items}' \leq \textit{max} \\
\#\textit{items} < \textit{max} \\
\textit{items}' = \textit{items}^\frown < \textit{msg}? > \\
\hline
\end{array}
$$

- The first two lines of the predicate indicate that we have a valid instance of the state schema **Buffer** both before and after the operation.
- The third line of the predicate is a pre-condition for the operation. It indicates that for the **Join** operation to be possible, the buffer must not be completely full.
- The last line of the predicate specifies the relationship between the buffer contents before and after the operation which is that the input message is already appended to the sequence of messages already in the buffer.

Formal Specification: Message Buffer (Continued)

Operation Schema: Leave

- The decoration ! denotes an output
- There is an implicit $\wedge$ between each line in the predicate section.
- Explanation for Leave Operation Predicate
  1. The first two lines of the predicate indicate that we have a valid instance of the state schema **Buffer** both before and after the operation.
  2. The third line of the predicate is a **pre-condition** for the operation. It indicates that for the **Leave** operation to be possible, the buffer must not be empty.
  3. The last line of the predicate specifies the relationship between the buffer contents before and after the operation. The output message is taken from the head of the sequence of messages in the buffer, leaving just the tail of the sequence of buffers.

Leave

$items, items' : \mathrm{seq}\, MSG$

$msg! : MSG$

$\#items \leq max$

$\#items' \leq max$

$\#items \neq \emptyset$

$items = < msg! > \frown items'$

Special States: Delta (Δ) and Initial State ($_{INIT}$)

1. Delta (Δ): To specify a **before** and **after** instance of the state schema for any operation.

$$\Delta Buffer$$
$$items, items' : \mathrm{seq}\, MSG$$
$$\#items \leq max$$
$$\#items' \leq max$$

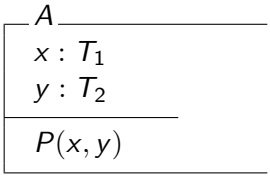
2. Initial State ($_{INIT}$): To specify a state when an instance of a state is first initialized.

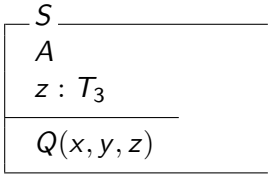
$$Buffer_{INIT}$$
$$Buffer$$
$$items = <>$$

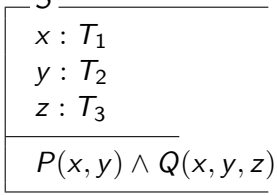
- Initially the buffer would be empty.
- Then, the operations of **Join** and **Leave** can occur whenever they are enabled.
- Operations are assumed to be atomic.
- At all times, an observer will notice that the state schema is satisfied.

Schema Inclusion

- Schema Inclusion is the act of including a schema in the declaration of another schema.
- It means the included schema has its declaration added to the new schema, and its predicate cojoined to the predicate of the new schema.
- The first "S" Schema is the **short form**, while the second "S" Schema is the **long form**.







Example: Schema Inclusion

Join

$\Delta Buffer$

$msg? : MSG$

$\#items < max$

$items' = items^\frown < msg? >$

Leave

$\Delta Buffer$

$msg! : MSG$

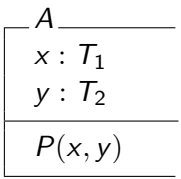
$items \neq \emptyset$

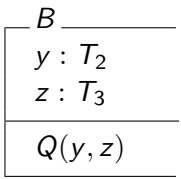
$items =< msg! > ^\frown items'$

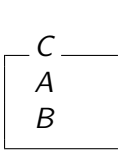
- We can include the $\Delta Buffer$ schema to both the Join and Leave operations.
- Explanation for Leave Operation Predicate
  1. The first line of the predicate is a **pre-condition** for the operation. It indicates that for the **Leave** operation to be possible, the buffer must not be empty.
  2. The last line of the predicate specifies the relationship between the buffer contents before and after the operation. The output message is taken from the head of the sequence of messages in the buffer, leaving just the tail of the sequence of buffers.

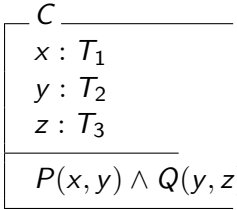
Merging Schemas

- **Type Compatability** is needed to merge schemas.
- In this case, the variable y is common between states A and B.
- We can simply merge the two types into a new state C without further specifying any new predicates.
- The full form of state C is also provided.



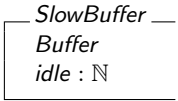


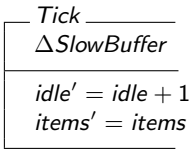


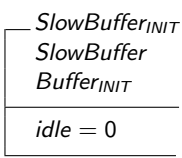


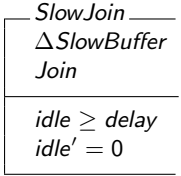
Extending Specifications: Slow Buffer and Slow Operations

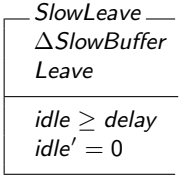
- Applying concepts such as **Schema Inclusion** and **Merging Schemas**, we can extend specifications similar to inheritance or creating specialized classes in Object-Oriented Programming.
- To demonstrate this, we will attempt to model a **Slow Buffer** which has a constant $delay : \mathbb{N}$ to simulate that each new message can only join the buffer after $delay$ seconds.











Extending Specifications: Slow Buffer and Slow Operations (Full Form)

SlowBuffer

$$items : \operatorname{seq} MSG$$
$$idle : \mathbb{N}$$
$$\#items \leq max$$

SlowBuffer$_{INIT}$

$$items : \operatorname{seq} MSG$$
$$idle : \mathbb{N}$$
$$\#items \leq max$$
$$items = <>$$
$$idle = 0$$

SlowJoin

$$items, items' : \operatorname{seq} MSG$$
$$idle, idle' : \mathbb{N} \quad msg? : MSG$$
$$\#items \leq max \land \#items' \leq max$$
$$\#items < max \land items' = items^\frown < msg? >$$
$$idle \geq delay \land idle' = 0$$

SlowLeave

$$items, items' : \operatorname{seq} MSG$$
$$idle, idle' : \mathbb{N}$$
$$msg! : MSG$$
$$\#items \leq max \land \#items' \leq max$$
$$items \neq \emptyset \land items = items'^\frown < msg? >$$
$$idle \geq delay \land idle' = 0$$

Tick

$$items : \operatorname{seq} MSG$$
$$idle : \mathbb{N}$$
$$\#items \leq max \land \#items' \leq max$$
$$idle' = idle + 1 \land items' = item$$

Reasoning About The Specification

- Suppose, we want to verify that message buffer specified has the **FIFO property**.
- We want to show that the messages leave the buffer in the same order they arrive.
- In this case, we introduce **auxillary sequences** `inhist` and `outhist` to record the history of the flow of messages into and out of the buffer.
- Create a new schema which includes the original Buffer and Operation schemas and extra information about the auxillary variables.
- When a message **joins** the buffer, it is also added to the `inhist` sequence.
- When a message **leaves** the buffer, it is added to the `outhist` sequence.

Recorded Buffer and Operations

RecordedBuffer

$Buffer$

$inhist : \mathrm{seq}\ MSG$

$outhist : \mathrm{seq}\ MSG$

RecordedJoin

$\Delta RecordedBuffer$

$Join$

$inhist' = inhist^\frown < msg? >$

$outhist' = outhist$

RecordedBuffer$_{INIT}$

$RecordedBuffer$

$Buffer_{INIT}$

$inhist = <>$

$outhist = <>$

RecordedLeave

$\Delta RecordedBuffer$

$Join$

$inhist' = inhist$

$outhist' = outhist^\frown < msg! >$

RecordedJoin: Expanded Schema

RecordedJoin

$items, items' : \operatorname{seq} MSG$

$inhist, inhist' : \operatorname{seq} MSG$

$outhist, outhist' : \operatorname{seq} MSG$

$msg? : MSG$

$\#items \leq max \land \#items' \leq max$

$\#items < max \land items' = items^\frown < msg? >$

$inhist' = inhist^\frown < msg? > outhist' = outhist$

Proving the FIFO Property

- How can we use the auxillary variables `inhist` and `outhist` to prove that the buffer satisifies the **FIFO property**?
- We can prove that the predicate $\forall\, RecordedBuffer \bullet inhist = outhist \frown items$ is true.
- Prove using structural induction.
  1. Initially `inhist = outhist = items = <>`, so the predicate is true for the initial state.
  2. Suppose the predicate is true, and RecordedJoin occurs. After the operation, $inhist' = inhist \frown < msg? > \land outhist' = outhist \land items' = items \frown < msg? >$
  3. Hence, $inhist \frown < msg? > = (outhist \frown items) \frown < msg? >$ $= outhist \frown (items \frown < msg? >) = outhist' \frown items'$
  4. Therefore, the predicate remains true.
- We can construct a similar argument that the operation **RecordedLeave** also preserves the predicate.

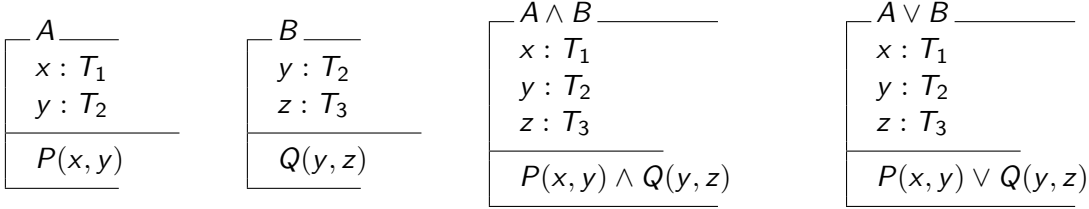
Conjunction of Schemas

- When using the conjunction ($\wedge$) operator on two schemas, it is equivalent to merging the two schemas.
- Suppose A and B are schemas
  - The declaration of $A \wedge B$ is the **union** of the declarations of A and B
  - The predicate of $A \wedge B$ is the **conjunction** of the predicates of A and B
- Examples
  1. $SlowRecordedBuffer \mathrel{\widehat=} SlowBuffer \wedge RecordedBuffer$
  2. $SlowRecordedBuffer_{INIT} \mathrel{\widehat=} SlowBuffer_{INIT} \wedge RecordedBuffer_{INIT}$
  3. $SlowRecordedJoin \mathrel{\widehat=} SlowJoin \wedge RecordedJoin$
- SlowRecordedBuffer Schema

$$\begin{array}{|l} \hline SlowRecordedBuffer \\ \hline SlowBuffer \\ RecordedBuffer \\ \hline \end{array}$$

Disjunction of Schemas

- Using the conjunction ($\wedge$) operator on two schemas yields a different result.
- Suppose A and B are schemas
  - The declaration of $A \vee B$ is the **union** of the declarations of A and B
  - The predicate of $A \vee B$ is the **disjunction** of the predicates of A and B



Disjunction of Schemas: Example

- Let $Flag ::= ok \mid error$ (The Flag type can be either 'ok' or 'error')
- Ξ State is another special state that is used for operations that access information in the state **without changing the state at all**.
- Example: $CompleteJoin \mathrel{\widehat=} JoinOk \lor JoinError$

JoinOk

$Join$

$flag! : Flag$

$flag! = ok$

JoinError

$\Xi Buffer$

$flag! : Flag$

$\#items = max \land flag! = error$

CompleteJoin

$\Delta Buffer$

$msg? : MSG;\ flag! : Flag$

$\#items < max \land items' = item^\frown < msg? > \land flag! = ok$

$\lor$

$\#items = max \land items' = items \land flag! = error$

Composition of Schemas

- Using the composition operator ($\fatsemi$) on two schemas is typically used to combine the effects of two operations.
- Example: $JoinLeave = Join \fatsemi Leave$
  - The pre-state of *Join* is the pre-state of $Join \fatsemi Leave$.
  - The post-state of *Join* is identified with the pre-state of *Leave* hidden within $Join \fatsemi Leave$.
  - The consequent post-state of *Leave* is the post-state of $Join \fatsemi Leave$.
- Convention: Hidden state is denoted with double prime (″).

JoinLeave

$$\Delta Buffer$$
$$msg?, msg! : MSG$$

$$\#items < max$$
$$\exists items'' : \mathrm{seq}\, MSG \bullet items'' = items^\frown < msg? > \wedge items'' = < msg! > ^\frown items'$$

Composition of Schemas in general

A

$x : T_1$
$y : T_2$

$P(x, y)$

AOP_1

δA
$t_3? : T_3;\ t_4! : T_4$

$Q_1(x, x', y, y', t_3?, t_4!)$

AOP_2

δA
$t_5? : T_5;\ t_6! : T_6$

$Q_2(x, x', y, y', t_5?, t_6!)$

$AOP_1 \fatsemi AOP_2$

δA
$t_3? : T_3;\ t_4! : T_4;\ t_5? : T_5;\ t_6! : T_6$

$\exists x'' : T_1;\ y'' : T_2 \bullet Q_1(x, x', y, y', t_3?, t_4!) \land Q_2(x, x', y, y', t_5?, t_6!)$

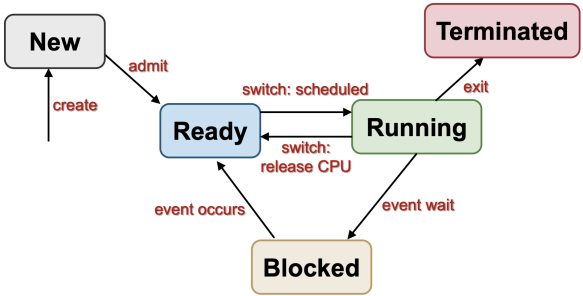
Table of Contents

Introduction to CSP

- **Communicating Sequential Processes (CSP)** is a formal language to describe patterns interaction between concurrent processes in computer systems.
- CSP was created by **Tony Hoare** to reason about the behaviour of concurrent systems.
- CSP provides **event based** notation primarily aimed at describing the sequencing of behaviour within a process and the synchronisation of behaviour (or communication) between processes.
- In CSP, there are two main fundamental concepts
  1. Process
  2. Event
- Events represent a co-operative synchronisation between process and environment.
- Both process and environment may control the behaviour of each other by enabling or refusing certain events or sequences of events.

Recap: Process State Model

- To gain a better understanding of **Communicating Sequential Processes**, revisit the first half of **CS2106: Operating Systems** where Processes, Concurrency, Synchronisation and Semaphores were covered.
- The Process State Model is shown here to help you visualize potential process behaviour that can be modelled using CSP.



Specifying a Process

- A process is determined or specified by what it can do.
- In other words, a process is defined by its behaviour, which are the events that we can observe.
- The perceived behaviour of a process will depend on the observer.
- In this course, we are mainly concerned with specifying the interaction between a system and its environment which is also the **external or visible behaviour**.

Concepts in CSP: Events

- A process engages in **events**.
- Each event is an atomic action.

Alphabet

The **set of events** a process can possibly engage in is the **alphabet** of the process.

- Example: Chocolate Vending Machine
- Events for a chocolate vending machine
  1. coin - insert a coin
  2. choc - extract a chocolate
- The alphabet of a chocolate vending machine is $\{coin, choc\}$

Concepts in CSP: Trace

Trace

A finite sequence of events

- A deterministic process is specified by the set of processes denoting its possible behaviour.
- Any execution of the process will be one of these sequences.
- Example: The traces of the chocolate vending machine are:
  - $\langle\rangle$
  - $\langle coin \rangle$
  - $\langle coin, choc \rangle$
  - $\langle coin, choc, coin \rangle$
  - ...
- Any execution of the process will be one of the above sequences or traces.
- If $s \frown t$ is a trace of a process, then s is also a trace of a process. This means that set of traces is prefixed closed.
  - Example: Let $s = \langle coin, choc \rangle, t = \langle coin \rangle$, then $s \frown t = \langle coin, choc, coin \rangle$.

Notations and Conventions

- **Events** are denoted in **lower case**
  - Example: x, y, z are variables that denote events.
- **Processes** are denoted in **upper case**
  - Example: X, Y, Z are variables that denote processes.
- The **alphabet** of a process P is denoted by αP
- The set of traces of P is denoted by `traces(P)`
- **Trace Notation**
  - If A is a set of events, then $\mathrm{seq}\, A$ denotes the set of all finite sequences of events from A.
  - In this scenario, α `P = A` and `trace(P) =` $\mathrm{seq}\, A$
  - Let $s, t : \mathrm{seq}\, A$, $s \frown t$ be the concatenation of s with t.
  - We define the relation $<=$ to be the **sequence prefix** of two sequences.

$$\begin{array}{|l} <= : \mathrm{seq}\, A \leftrightarrow \mathrm{seq}\, A \\ \hline s <= t \Leftrightarrow \exists u : \mathrm{seq}\, A \bullet s \frown u = t \end{array}$$

  - $s^n = s \frown s \frown s \frown \ldots \frown s$ denotes the event s concatenated with itself n times.

Examples of Process Specification using CSP

1. Let $STOP_A$ be the process with alphabet A that can do nothing.
   - $traces(STOP_A) = \{\langle\rangle\}$
2. Let CLOCK be the process with $\alpha CLOCK = tick$ which can 'tick' at any time.
   - $traces(Clock) = tick^*$ where $* \geq 0$
3. Let VM be the process with $\alpha VM = \{coin, choc\}$ which repeatedly supplies a chocolate after a coin has been inserted.
   - $traces(VM) = \{s : \mathrm{seq}\{coin, choc\} \mid \exists\, n : \mathbb{N} \bullet s <= \langle coin, choc\rangle^n\}$
   - Note: $<=$ is the sequence prefix relation from the previous slide.
4. Let WALK be a one-dimensional random walk process with $\alpha WALK = \{left, right\}$.
   - $traces(WALK) = (left \lor right)^*$ where $* \geq 0$
5. Let LIFE be the process with $\alpha LIFE = \{beat\}$ which can stop (die) at any time.
   - $traces(LIFE) = beat^*$ where $* \geq 0$

CSP Syntax

1. Prefix: $a \rightarrow P$
2. Sequential Composition: $P;\ Q$
3. Parallel Composition (Synchronous): $P \,|[\, X \,]|\, Q$
4. Interleaving (Asynchronous): $P \,|||\, Q$
5. Choice: $a \rightarrow P \,\Box\, b \rightarrow Q$
6. Interrupt Process: $P \,\nabla\, e \rightarrow Q$

[0] Note that in the `zed-csp` package, the interrupt symbol is $\triangle$.

Prefix

- A process which may participate in **event a** then act according to **process description P** is written as: `a -> P`.
- **a** is the event prefix to **P**.
- The **event a** is initially enabled by the process and occurs as soon as it is requested by its environment. All other events are refused initially.
- The **event a** is sometimes referred to as the guard of the process.
- Examples
  1. $VMU = coin \rightarrow STOP$
  2. $SHORTLIFE = (beat \rightarrow (beat \rightarrow STOP)) = beat \rightarrow beat \rightarrow STOP$
  3. $VMS = coin \rightarrow choc \rightarrow STOP$

Sequential Composition ($P;\ Q$)

- Let $\checkmark$ be the Termination event.
- The process which may only terminate is written as $SKIP$.
- Let $SKIP = \checkmark \rightarrow STOP$.
- The sequential composition of processes P and Q, written as $P;\ Q$, acts as P until P terminates by communicating $\checkmark$ and then proceeds to act as Q.

Parallel Composition

- The parallel composition of **processes P and Q** synchronised on the **event set X** is written as $P \,|[\, X \,]|\, Q$.
- No event from X may occur in $P \,|[\, X \,]|\, Q$ unless **jointly enabled** by both P and Q.
- When events from X occur, they occur in both P and Q simultaneously, and are referred to as **synchronisations**.
- Events **not from X** may occur in either P or Q seperately **but not jointly**.
- Example: $(a \rightarrow P) \,|[\, a \,]|\, (c \rightarrow a \rightarrow Q)$
  - All **a** events must be Synchronous between the two processes.
- Often, it is simply written as $P \parallel Q$ where the common event set $X = \alpha P \cap \alpha Q$ is omitted.
  - When $P \parallel Q$ is given, we still know that all common events in $X = \alpha P \cap \alpha Q$ must be synchronous between P and Q.

Interleaving

- $P \mid\mid\mid Q$ denotes an asynchronous parallel composition between two processes **P** and **Q**.
- Both components **P** and **Q execute concurrently** without any synchronisation.
- Example: $((a \rightarrow P) \mid\mid\mid (c \rightarrow a \rightarrow Q))$
  - One possible trace is $\langle c, a, a \rangle$, after which the process acts as $P \mid\mid\mid Q$
    1. c from $c \rightarrow a \rightarrow Q$ is engaged, leaving us with $((a \rightarrow P) \mid\mid\mid (a \rightarrow Q))$
    2. a from $a \rightarrow P$ is engaged, leaving us with $(P \mid\mid\mid (a \rightarrow Q))$
    3. a from $a \rightarrow Q$ is engaged, leaving us with $P \mid\mid\mid Q$
  - Another possible trace is $\langle a, c, a \rangle$, after which the process acts as $P \mid\mid\mid Q$
    1. a from $a \rightarrow P$ is engaged, leaving us with $(P \mid\mid\mid (c \rightarrow a \rightarrow Q))$
    2. c from $c \rightarrow a \rightarrow Q$ is engaged, leaving us with $(P \mid\mid\mid (a \rightarrow Q))$
    3. a from $a \rightarrow Q$ is engaged, leaving us with $P \mid\mid\mid Q$

Choice

- In a general choice, $(a \rightarrow P) \,\square\, (b \rightarrow Q)$, the process begins with both events **a** and **b** enabled.
- The subsequent behaviour depends on the event which occured.
  - If the event which occured is **a**, the process will act as **P** afterwards.
  - If the event which occured is **b**, the process will act as **Q** afterwards.
- There are other types of choices, namely external or internal choice in the CSP syntax.
- For most cases, general choice is sufficient, hence in this module, we focus on general choice only.
- Example: $(a \rightarrow P) \,\square\, (c \rightarrow a \rightarrow Q)$
  - If the first event is **a**, after which the process acts as P.
  - If the first event is **c**, after which the process acts as $a \rightarrow Q$.

Interrupt

- The interrupt process $P \bigtriangledown e \rightarrow Q$ behaves as **process P** until the first occurrence of **event e** which then the control passes to **process Q**.
- When coding the specification, the keyword **interrupt** is used instead of the symbol $\bigtriangledown$.
- For the System process, the first event can be a routine or an exception.
- After that, it still behaves as a System process.

```
1 Err() = exception -> Err();
2 Routine() = routine -> Routine();
3 ExceptionHandling() = Routine() interrupt exception -> ExceptionHandling();
4 System = Err() || ExceptionHandling();
```

[0] Note that in the zed-csp package, the interrupt symbol is $\bigtriangleup$.

Concurrency Example #1

We are given the following system specification in CSP

```
1 VMC = coin -> ((choc -> VMC)[](bisc -> VMC));
2 CHOCLOV = choc -> CHOCLOV [] coin -> choc -> CHOCLOV;
3 #alphabet VMC{coin, choc, bisc};
4 #alphabet VMC{coin, choc, bisc};
5 System = VMC || CHOCLOV;
```

- Semi-colon marks the end of each statement.
- When coding process specifications in CSP, we use the keyword **#alphabet**, instead of the symbol α.
- The only possible trace for this example is $\langle coin, choc \rangle^n$ for $n : \mathbb{N}_1$ after which the system acts as $VMC \parallel CHOCLOV$.
  - As defined in lines 3 and 4 of the specification, the common events are the alphabets of VMC and CHOCLOV which are **coin**, **choc** and **bisc**.
  - However, the process **CHOCLOV** does not have the event **bisc**.
  - In VMC, the **coin** event has to be engaged first before we can engage $choc \rightarrow VMC$
  - Hence, both VMC and CHOCLOV engages in the **coin** event first then the **choc** event before acting as $VMC \parallel CHOCLOV$ again.

Concurrency Example #2

We are given the same specification as the previous slide in CSP except that we now do not define the alphabet for the **VMC** and **CHOCLOV** processes.

```
1 VMC = coin -> ((choc -> VMC)[](bisc -> VMC));
2 CHOCLOV = choc -> CHOCLOV [] coin -> choc -> CHOCLOV;
3 System = VMC || CHOCLOV;
```

- If we did not explicity define the alphabet for each process, it can be **auto inferred**.
  - $\alpha VMC = \{coin, choc, bisc\}$
  - $\alpha CHOCLOV = \{coin, choc\}$
- In this specification the system may deadlock with the following trace $\langle coin, bisc \rangle$.
  - After $\langle coin, bisc \rangle$, no event is possible.
  - This is because now the **bisc** event is not common to both **VMC** and **CHOCLOV** processes.
  - Hence, the **bisc** event can occur seperately.
  - After $\langle coin, bisc \rangle$ has occurred, the system would be stuck at $VMC \parallel choc \rightarrow CHOCLOV$.
  - Since **coin** and **choc** are common events, neither events can be engaged synchronously as **coin** is a prefix for **choc**.

Concurrency Example #3

We are given the following system specification in CSP

```
1 VMH = on -> coin -> choc -> off -> VMH;
2 CUST = on -> ((coin -> bisc -> CUST) [] (curse -> coin -> choc -> CUST));
3 System = VMH || CUST;
```

- The common events between **VMH** and **CUST** are **on**, **coin** and **choc** and they must occur synchronously in the two processes.
- $\langle on, curse, coin, choc, off \rangle$ is a possible trace.
  - After the trace, the process will still behave as a System process.
- Deadlocks can occur with the following trace $\langle on, coin, bisc \rangle$
  - The system will be stuck at $choc \rightarrow off \rightarrow VMH \parallel CUST$ and no event can be engaged.

Concurrency Example #4

We are given the following system specification in CSP

```
SLOWALK = left -> rest -> SLOWALK [] right -> rest -> SLOWALK;
SLOCLIMB = up -> rest -> SLOCLIMB [] down -> SLOCLIMB;
System = SLOWALK || SLOCLIMB;
```

Are the following traces possible?

1. $\langle up, rest \rangle$
   - No. The common event between **SLOWALK** and **SLOCLIMB** is **rest**.
   - $\langle up, left, rest \rangle$ and $\langle up, right, rest \rangle$ are possible traces.
2. $\langle \ldots, up, rest \rangle$ where up may not the first event.
   - Yes. For example $\langle left, up, rest \rangle$ and $\langle right, up, rest \rangle$.

Laws for Concurrency

- Law 1: $P \parallel Q = Q \parallel P$
- Law 2: $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$
- Law 3: $P \parallel STOP_{\alpha P} = STOP_{\alpha P}$

Let...

1. $a \in (\alpha P - \alpha Q)$
2. $b \in (\alpha Q - \alpha P)$
3. $\{c, d\} \subseteq (\alpha P \cap \alpha Q)$

- Law 4A: $(c \rightarrow P) \parallel (c \rightarrow Q) = c \rightarrow (P \parallel Q)$
- Law 4B: $(c \rightarrow P) \parallel (d \rightarrow Q) = STOP$ if $c \neq d$
- Law 5A: $(a \rightarrow P) \parallel (c \rightarrow Q) = a \rightarrow (P \parallel (c \rightarrow Q))$
- Law 5A: $(c \rightarrow P) \parallel (b \rightarrow Q) = b \rightarrow ((c \rightarrow P) \parallel Q)$
- Law 6: $(a \rightarrow P) \parallel (b \rightarrow Q) = a \rightarrow (P \parallel (b \rightarrow Q)) \Box b \rightarrow ((a \rightarrow P) \parallel Q)$

Channel

- Processes may communicate through channels.
- A channel is like a message buffer for one process to send a value to another process.
- A channel event is written as one of the following forms:

Form	Description
$c!n$	Channel Output. This event occurs when a process writes n (a value) to the tail of channel c's buffer
$c?n$	Channel Input. This event occurs when a process reads a value from the head of channel c's buffer to a local variable n
$c.n$	Channel output and its matching channel input are engaged together by two processes.

Channel Example

Suppose we are given the following specification in CSP.

```
channel c 1; // Channel with buffer size = 1
Sender(i) = c!i -> Sender(i);
Receiver() = c?x -> a.x -> Receiver();
System() = Sender(5) ||| Receiver();
```

- Note: A process can have optional parameters, eg: Sender(i)
- The first event must be c!5 since c's buffer is empty.
- The second event must be c?5 since c's buffer size is 1.
- The third event can either be c!5 or a.5

Channel Example: Synchronous Buffer

Suppose we are given the following specification in CSP.

```
channel c 0; // Synchronous Buffer
Sender(i) = c!i -> Sender(i);
Receiver() = c?x -> a.x -> Receiver();
System() = Sender(5) ||| Receiver();
```

- Note: A synchronous buffer is defined by setting the buffer size to 0.
- The first event must be c.5, since the sender must write to the c's buffer and the reciever must read from c's buffer simultaneously.
- The second event must be a.5

CSP Model Checkers: Automatic Reasoning

- A CSP Model Checker can check if a property is always satisfied.
- In the next part, we will explore how we can verify certain properties of our process model using PAT (CSP#).
- Suppose we are given the specification of the **Dining Philosophers Problem**.

```
1 #define N 2;
2 Phil(i) = get.i.(i+1)%N -> get.i.i -> eat.i -> put.i.(i+1)%N -> put.i.i ->
      Phil(i);
3 Fork(x) = get.x.x -> put.x.x -> Fork(x) [] get.(x-1)%N.x -> put.(x-1)%N.x ->
       Fork(x);
4 College() = ||x:{0..N-1}@(Phil(x)||Fork(x));
5 #assert College() deadlockfree; // Check if a property is satisifed.
```

- $\| \, x : \{1 \ldots n\} @ P(x)$ is equivalent to $P(1) \parallel \ldots \parallel P(n)$
- `#assert College() deadlockfree` is the property we want to verify.
- If the property isn't always True, the Model Checker gives a counter example: $\langle get.1.0, get.0.1 \rangle$

Table of Contents

Introduction to Process Analysis Toolkit (PAT)

- The Process Analysis Toolkit is a self-contained framework to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains.
- In PAT 3.5, there are 11 developed modules provided to model a variety of different systems ranging from **Communicating Sequential Processes (CSP) Module**, **Real-Time System Module** to the **Web Service Module**.
- In this module, we will be focusing on the **Communicating Sequential Programs (CSP#) module**

Introduction to PAT's CSP#

- PAT's CSP# module supports a rich modeling language named CSP#(pronounced 'CSP sharp', short for Communicating Sequential Programs)
- CSP# combines high-level modeling operators like (conditional or non-deterministic) choices, interrupt, (alphabetized) parallel composition, interleaving, hiding, asynchronous message passing channel, etc., with programmer-favored low-level constructs like variables, arrays, if-then-else, while, etc ...
- CSP# offers great flexibility on how to model systems. For instance, communication among processes can be either based on shared memory (using global variables) or message passing (using asynchronous message passing or CSP-style multi-party barrier synchronization).
- The high-level operators are based on the classic process algebra Communicating Sequential Processes (CSP).
- The design principle for CSP# is to maximally keep the original CSP as a sub-language of CSP#, whilst offering a connection to the data states and executable data operations.

Operational Semantics

- Earlier, we want a way to automatically reason if a model satisfies a certain property.
- Hence, we will need to study **Operational Semantics** to understand how a process transitions from one state to another during the execution of CSP Specification.
- **Operational Semantics** tell us at given a system state, what are the possible actions the system can perform and what are the outcomes (next state)?
  - Example: $P \xrightarrow{a} Q$
  - The above is read as: If process P engages event a, it will become process Q.
  - In most cases the states are the process and actions are the events.
  - We will see in later examples what are states and actions.
- Operational Semantics can be presented using a set of inference rules in the following form similar to the philosophy of logic.

$$\frac{\textit{Premises}}{\textit{Conclusion}}$$

Operational Semantics: Primitives

- STOP (A process that does nothing)
- SKIP

$$\frac{}{SKIP \xrightarrow{\checkmark} STOP}$$

SKIP can only engage the **termination event**, afterwards it becomes STOP.

- Prefixing

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

$a \rightarrow P$ can only engage event **a**, afterwards it becomes process **P**.

Operational Semantics

General Choice

- If P is choosen

$$\frac{P \xrightarrow{a} P'}{(P \,\square\, Q) \xrightarrow{a} P'}$$

- If Q is chosen

$$\frac{Q \xrightarrow{a} Q'}{(P \,\square\, Q) \xrightarrow{a} Q'}$$

Sequential Composition

- In process P;Q, P takes control first and Q starts only when P has finished.
- Let ✓ be the termination event.

$$\frac{P \xrightarrow{a} P'}{(P;\ Q) \xrightarrow{a} (P';\ Q)}$$

$$\frac{P \xrightarrow{\checkmark} P'}{(P \,\square\, Q) \xrightarrow{\checkmark} Q}$$

Interrupt

- In process P ∇ Q, whenever an event is engaged by Q, P is interrupted and the control is transferred to Q.

$$\frac{P \xrightarrow{a} P'}{(P \nabla Q) \xrightarrow{a} (P' \nabla Q)}$$

$$\frac{Q \xrightarrow{a} Q'}{(P \nabla Q) \xrightarrow{a} Q}$$

Operational Semantics: Example #1

- Let $VMS = coin \rightarrow (choc \rightarrow VMS \ \Box \ bisc \rightarrow VMS)$

Operation Semantics

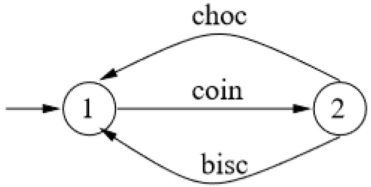
Step 1. $VMS \xrightarrow{coin} (choc \rightarrow VMS \ \Box \ bisc \rightarrow VMS)$

Step 2. $(choc \rightarrow VMS \ \Box \ bisc \rightarrow VMS) \xrightarrow{choc} VMS$

Step 2. $(choc \rightarrow VMS \ \Box \ bisc \rightarrow VMS) \xrightarrow{bisc} VMS$

Labelled Transition System (LTS)

- A **Labelled Transition System** contains a set of states, an initial state (where the system starts from) and a labelled transition relation.



- Let $VMS = coin \rightarrow (choc \rightarrow VMS \ \square \ bisc \rightarrow VMS)$
- State 1 represents the process VMS.
- State 2 represents the process $(choc \rightarrow VMS \ \square \ bisc \rightarrow VMS)$

[1] The Labelled Transition System is a directed graph

Operational Semantics

Interleaving

- In process $P \,|||\, Q$, P and Q behaves independently.
- The exception is the termination, hence assume a is not $\checkmark$.

$$\frac{P \xrightarrow{a} P'}{(P \,|||\, Q) \xrightarrow{a} (P' \,|||\, Q)}$$

$$\frac{Q \xrightarrow{a} Q'}{(P \,|||\, Q) \xrightarrow{a} (P \,|||\, Q')}$$

Synchronization

- In process $P \,|[\, X \,]|\, Q$, no event from X may occur unless jointly by both P and Q.
- When events from X do occur, they occur in P and Q simultaneously.

$$\frac{P \xrightarrow{a} P', a \notin X}{(P \,|[\, X \,]|\, Q) \xrightarrow{a} (P' \,|[\, X \,]|\, Q)}$$

$$\frac{Q \xrightarrow{a} Q', a \notin X}{(P \,|[\, X \,]|\, Q) \xrightarrow{a} (P \,|[\, X \,]|\, Q')}$$

$$\frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q', a \in X}{(P \,|[\, X \,]|\, Q) \xrightarrow{a} (P' \,|[\, X \,]|\, Q')}$$

Operational Semantics: Example #2

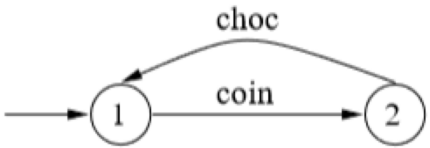
Given the process $a \rightarrow P \,|[\, a \,]|\, (c \rightarrow a \rightarrow Q)$

1. $(a \rightarrow P \,|[\, a \,]|\, (c \rightarrow a \rightarrow Q)) \xrightarrow{c} (a \rightarrow P \,|[\, a \,]|\, (a \rightarrow Q))$
   Only event **c** can be engaged at first as **a** is a common event in both $a \rightarrow P$ and $c \rightarrow a \rightarrow Q$.
2. $(a \rightarrow P \,|[\, a \,]|\, (a \rightarrow Q)) \xrightarrow{a} (P \,|[\, a \,]|\, Q)$
   Engage the common event **a** on both $a \rightarrow P$ and $a \rightarrow Q$.

Operational Semantics: Example #3

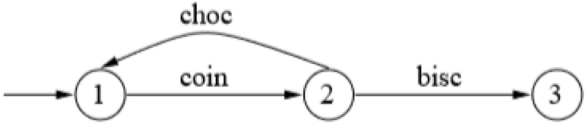
- $VMC = coin \rightarrow (choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC)$
- $CHOCLOV = choc \rightarrow CHOCLOV \ \Box \ coin \rightarrow choc \rightarrow CHOCLOV$

1. How would the process $VMC \,||[\,A\,]||\, CHOCLOV$ behave when $A = \{coin, choc, bisc\}$
   - Step 1:
     $VMC \,||[\,A\,]||\, CHOCLOV \xrightarrow{coin} (choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC) \,||[\,A\,]||\, (choc \rightarrow CHOCLOV)$
   - Step 2:
     $(choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC) \,||[\,A\,]||\, (choc \rightarrow CHOCLOV) \xrightarrow{choc} VMC \,||[\,A\,]||\, CHOCLOV$



Operational Semantics Example: #4

- $VMC = coin \rightarrow (choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC)$
- $CHOCLOV = choc \rightarrow CHOCLOV \ \Box \ coin \rightarrow choc \rightarrow CHOCLOV$
2. How would the process $VMC \ |[\{coin, choc\}]| \ CHOCLOV$ or equivalently $VMC \parallel CHOCLOV$ behave?
   - Step 1: $VMC \parallel CHOCLOV \xrightarrow{coin} (choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC) \parallel (choc \rightarrow CHOCLOV)$
   - Step 2:
     $(choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC) \ |[A]| \ (choc \rightarrow CHOCLOV) \xrightarrow{choc} VMC \ |[A]| \ CHOCLOV$
   - Step 2: $(choc \rightarrow VMC \ \Box \ bisc \rightarrow VMC) \ |[A]| \ (choc \rightarrow CHOCLOV) \xrightarrow{choc}$
     $VMC \ |[A]| \ (choc \rightarrow CHOCLOV)$



Case Study: Dining Philosophers

1. Specify the dining philosophers

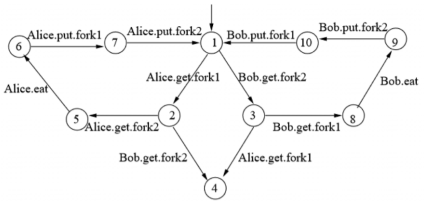
```
1 Alice = Alice.get.fork1 -> Alice.get.fork2 -> Alice.eat -> Alice.put.
      fork1 -> Alice.put.fork2 -> Alice
2 Bob = Bob.get.fork1 -> Bob.get.fork2 -> Bob.eat -> Bob.put.fork1 -> Bob.
      put.fork2 -> Bob
3 Fork1 = (Alice.get.fork1 -> Alice.put.fork1 -> Fork1) [] (Bob.get.fork1
      -> Bob.put.fork1 -> Fork1)
4 Fork2 = (Alice.get.fork2 -> Alice.put.fork2 -> Fork2) [] (Bob.get.fork2
      -> Bob.put.fork2 -> Fork2)
5 College = Alice || Bob || Fork1 || Fork2
```

2. Get the alphabets of each process
   - $\alpha Alice = \{Alice.get.fork1, Alice.get.fork2, Alice.eat, Alice.put.fork1, Alice.put.fork2\}$
   - $\alpha Bob = \{Bob.get.fork1, Bob.get.fork2, Bob.eat, Bob.put.fork1, Bob.put.fork2\}$
   - $\alpha Fork1 = \{Alice.get.fork1, Alice.put.fork1, Bob.get.fork1, Bob.put.fork1\}$
   - $\alpha Fork2 = \{Alice.get.fork2, Alice.put.fork2, Bob.get.fork2, Bob.put.fork2\}$

Case Study: Dining Philosophers

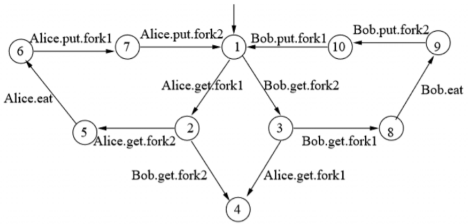
3. Apply the operational semantics rule (one at a time) to build the Labelled Transition System.
   - Alice can perform Alice.get.fork1
   - Bob can perform Bob.get.Fork2
   - Fork1 can perform Alice.get.fork1 or Bob.get.fork1
   - Fork2 can perform Alice.get.fork2 or Bob.get.Fork2
   - By rule syn3, College can perform either Alice.get.fork1 or Bob.get.fork2, and then a state of the form.

$$\cdots \parallel \cdots \parallel \cdots \parallel \cdots$$



Case Study: Dining Philosophers

4. Analyze the Labelled Transition system
   - Is the system deadlock-free?
   - Will Alice or Bob starve to death?



Safety

- Safety means **something bad never happens**.
- Examples
  1. deadlock-freeness

     ```
     #assert College() deadlockfree;
     ```

     The system never deadlocks
  2. invariant

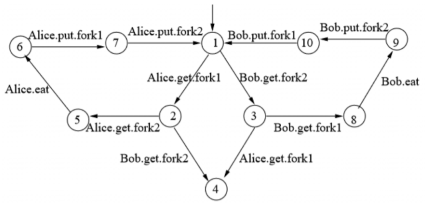
     ```
     #assert Bank() |= [] Value >= Debit;
     ```

     The savings of a bank account must always be non-negative.

[1] '[]' signifies 'always' in Linear Temporal Logic.
[2] '|=' represents 'satisfisfaction' in Linear Temporal Logic.

Verifying Safety

- To verify safety, perform **reachability analysis** on the **Labelled Transition System**.
- A counterexample to the safety property is a finite execution which leads to a bad state.
- Perform either **Depth First Search (DFS)** or **Breadth First Search (BFS)** to search all reachable states for a 'bad' one.
- Example:



1. Depth First Search: $1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 1 \rightarrow$ backtrack $\rightarrow 4 \rightarrow$ FOUND!
2. Breadth First Search: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 4 \rightarrow$ FOUND!

[1] State 4 is the bad state as there is no outgoing edges.

Applications of Safety Verification

Many properties can be formulated as a safety property and solved using **reachability analysis**.

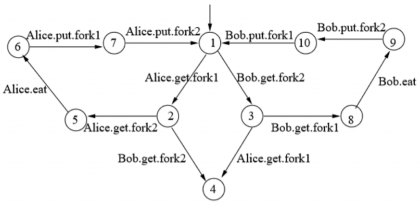
1. Mutual Exclusion: []!(more than one process accessing the critical section)
   - There will never be more than one process accessing the critical section.
2. Security: [](only the authorized user can access the information)
   - It is always the case that only the authorized user can access the information.
3. Program Analysis
   - Arrays are always bounded.
   - Pointers are always non-null
   - etc...

Liveness

- Liveness means **something good eventually happens**.
- Examples
  1. A program is eventually terminating.
  2. A file writer is eventually closed.
  3. Both Alice and Bob eventually get to eat.

Verifying Liveness

- To verify liveness, perform **loop searching** on the **Labelled Transition System**.
- A counterexample to a liveness property is an infinite system execution during which the 'good' thing never happens.
  - Example: An infinite loop fails the property that the program is eventually terminating.
- We can search through the Labelled Transition System for a bad loop using **Nested Depth First Search** or **Strongly Connected Component based Search**
- Example



Assertion: Alice will always eventually eat. (**False**)

```
1 #assert College() |= Alice.eat
```

Counterexamples

- $\langle Alice.get.fork1, Bob.get.fork2 \rangle$
- $< Bob.get.fork2 \rightarrow Bob.get.fork1 \rightarrow Bob.eat \rightarrow Bob.put.fork2, \rightarrow Bob.put.fork1 >^*$

CSP#: Extending CSP

- The original CSP has no shared variables, arrays, etc...
- PAT (CSP#) extends it with data operations.
- Hence, the operational semantics must be updated to support data operations.

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \,\square\, Q) \xrightarrow{x} -> (V', P')} \qquad \frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \,\square\, Q) \xrightarrow{x} -> (V', Q')}$$

Example

var x = 0;
$P = (a\{x = 1\} \rightarrow P) \,\square\, (b \rightarrow Skip)$;
P (with valuation var x = 0) $\xrightarrow{a}$ P (with valuation var x = 1)
P (with valuation var x = 0) $\xrightarrow{b}$ Skip (with valuation var x = 1)

CSP#: Global Definition

- Constants

```
#define max 5;
```

- Enumerations

```
enum {red, blue, green};
// Syntactic Sugar for the following
#define red 0;
#define blue 1;
#define green 2;
```

Note

- The constant value can only be of **integer** or **boolean** value.
- **#define** is a keyword used for multiple purposes. Here it defines a global constant.
- (;) **semi-colon** marks the end of the 'sentence'.

CSP#: Global Definition

- Variables

```
var knight = 0;
```

- Arrays

```
// A fixed sized array may be defined as follows:
var board = [3, 5, 6, 0, 2, 7, 8, 4, 1];
// If we do not specify the elements in an array,
// all elements in array are initialized to 0
var leader[3]; // Array of size 3.
var matrix[3][2] // Multi-dimensional array (internally an array of 6)
```

Note

- Multi-dimensional arrays are internally converted to one-dimension.
- The **var** keyword is used to defined variables.
- The scope of these variables are global if they are not within an event or process.

CSP#: Global Definition

- Often, it is desirable to provide the range of the variables / arrays explicity by giving the lower bound or upper bound or both.

```
var knight:{0..} = 0;
var board:{0..10} = [3, 5, 6, 0, 2, 7, 8, 4, 1];
```

- Array Initialization: To ease modelling, PAT supports fast array initialization using the following syntax

```
#define N 2;
// Initialized array with syntax shortcuts
var array = [1(2), 3..6, 7(N * 2), 12..10];
// The above is the same as the following
var array = [1, 1, 3, 4, 5, 6, 7, 7, 7, 7, 12, 11, 10];
```

CSP#: Global Definition

- **Macro**
  - The keyword **#define** may be used to define macros.

```
1 #define goal x == 0;
```

Explanation

- goal is the name of the macro
- x == 0 is what the goal means.

  - We can use macro to define system properties.

```
1 #assert System() reaches goal;
```

  - We can also use macro to define processes

```
1 if (goal) { P } else { Q };
```

Explanation

If the value of x is 0 then do P else do Q.

CSP#: Global Definition

- PAT only supports integer, boolean and integer arrays for the purpose of **efficient verification**.
- However, advanced data structures (eg: Stack, Queue, Hashtable, etc...) are necessary for some models.
- To support arbitrary data structures, PAT provides an interface to create user defined data types by inheriting an **abstract class** `ExpressionValue` using the **C# library**.
- For more details:
  - PAT User Manual: Section 2.5.2 User Defined Data Types

CSP#: Process Definition

- Event Prefixing
  1. Basic Form

```
1 e -> p;
2 VM() = coin -> coffee -> VM();
```

  2. Compound Form
     - For example in `x.exp1.exp2`, `x` is the event name and `exp1` and `exp2` are expressions.
     - Each expression corresponds to a variable (eg: process parameters, channel input variables or global variables).

```
1 #define N 2;
2 // Dining Philosophers Example
3 Phil(i) = get.i.(i + 1) % N -> Rest();
```

CSP#: Process Definition

- Statement Block inside Events (aka Data Operations)
  - An event can be attached with assignments which update global or local variables.
  - Process arguments and channel inputs can only used without being updated.
  - Semi-colons (;) mark the end of a statement in C# or end of a sentence in CSP#.

```
var array = [0, 2, 4, 7, 1, 3];
var maxi = -1;
P() = findmax {
    var index = 0;
    while (index < 6) {
        if (maxi < array[index]) { maxi = array[index]; }
        index = index + 1;
    };
} -> Skip;
```

Note

- P() = ... is equivalent to defining process P without any process parameters.

CSP#: Process Definition

- **Conditional Choice:** A choice may depend on a Boolean expression which in turn depends on the valuation of the variables.

```
var x = 1;
Init = []i{1,2}@set.i{x = i} -> Skip;
P = if (x == 1) { a -> Stop } else { b -> Stop };
System = Init;P; // Sequential composition of two processes
```

- **Guarded Process:** A guarded process only executes when its guard condition is satisfied.

```
var x = 1;
Init = []i{1,2}@set.i{x = i} -> Skip;
P = [x == 1] a -> Stop [] [x != 1] b -> Stop;
System = Init;P; // Sequential composition of two processes
```

Note

- `[]i:{1, 2}` means choice for variable i which can be either 1 or 2.
- In both examples, Process P behaves differently depending of the value of variable x.
- Both conditional choice and guarded process can produce the same effect.

CSP#: Process Definition

- **Guarded Process:** A guarded process only executes when its guard condition is satisfied.
- In the example, the trace $\langle b, b, a \rangle$ is possible but $\langle b, b, b, a \rangle$ is not possible.

```
1 var x = 0;
2 P = [x < 4] b{x = x + 1;} -> P;
3 aSys = [x == 2] a -> Stop ||| P;
```

CSP#: Process Definition

- **Atomic Process:**
  - The keyword **atomic** is used to indicate that a process is of **higher priority**.
  - This means if the atomic process has an enabled event, the event will execute before any events from non-atomic processes.

```
channel ch 0;
P = atomic { a -> ch!0 -> b -> Skip };
Q = atomic { d -> ch?0 -> e -> f -> Skip };
W = g -> Skip;
System = P ||| Q ||| W;
```

  - In the example above, processes P and Q are both atomic processes, while process W is not.
  - The expected behaviour is that processes P and Q will interleave each other (only synchronised on ch.0), whereas W will execute only after event b and f have occured.

Note

Since the channel size is 0, processes P and Q have to synchronise at the channel events.

CSP#: Assertions

- **Deadlock-freeness:** The following assertion asks if process P() is deadlock-free or not.
  - A deadlock state is a state with no further move, except for successfully terminated state.

```
P = a -> Skip;
#assert P deadlockfree; // True
```

- **Reachability:** The following assertion asks whether process P() can reach a state at which *some given condition is satisified.*
  - In this example, the assertion is True because it can reach a state where `x < 0`.

```
var x = 0;
P() = add{x = x + 1;} -> P() [] minus{x = x - 1;} -> P();
#define goal x < 0;
#assert P() reaches goal; // Note: goal condition must be a macro
```

CSP#: Assertions

The following coin exchanging example shows how to minimize the number of coins during reachability search.

```
var x = 0;
var weight = 0;
P() = if (x <= 14) {
    coin1{x = x + 1; weight = weight + 1;} -> P();
    [] coin2{x = x + 2; weight = weight + 1;} -> P();
    [] coin5{x = x + 5; weight = weight + 1;} -> P();
};
#define goal x == 14;
#assert P() reaches goal with min(weight);
```

CSP#: Assertions and Linear Temporal Logic

- **Linear Temporal Logic (LTL)** is a formalism used for specifying and reasoning about the behavior of systems over time.
- It extends propositional logic by introducing temporal operators that describe how properties of a system evolve over time, making it suitable for reasoning about sequences of states in a system.
- In LTL, time is viewed as a linear sequence of discrete points, and temporal operators allow the expression of future and current behaviors in a system.

CSP#: Assertions and Linear Temporal Logic

Let ϕ and ψ be LTL formulaes.

Operator	Usage	Name	Explanation
X	**X** ϕ	Next	ϕ has to hold at the next state
G	**G** ϕ	Globally	ϕ has to hold on the entire subsequent path
F	**F** ϕ	Finally	ϕ eventually has to hold somewhere on the subsequent path
U	ψ **U** ϕ	Until	ϕ holds at the current or a future position, and ψ has to hold until that position. At that position ψ does not have to hold anymore.
R	ψ **R** ϕ	Release	ϕ is true until the first position in which ψ is true, or forever if such a position does not exist.

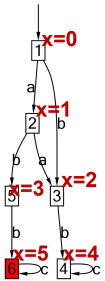
CSP#: Assertions and Linear Temporal Logic

- The LTL assertion is true if and only if every execution (**trace**) of the system satisfies the formula F, where F is an LTL formula whose syntax is defined as the following rules.
  - F = e | prop | [] F | <> F | X F | F1 U F2 | F1 R F2
  - Notations
    1. e is an event
    2. prop is a predefined propositional
    3. [] reads as "always" (or 'G' in CSP#)
    4. <> reads as "eventually" (or 'F' in CSP#)
    5. X reads as "next"
    6. U reads as "until"
    7. R reads as "release"

```
1 #assert P() |= F;
```

CSP#: Assertions and Linear Temporal Logic

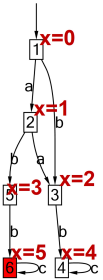
```
var x = 0;
P = [x < 2] a{x = x + 1} -> P
    [][x < 4] b {x = x + 2} -> P
    [][x >= 4] c -> P;
#define ge2 x >= 2;
#define lt2 x < 2;
```




```
#assert P deadlockfree; // Valid

#assert P |= lt2; // First state, x < 2?
                  // Yes

#assert P |= !c; // Init event is not c?
                 // Yes

#assert P |= X (a || b);
// Next event a or b? Yes!

#assert P |= [] ge2; // Always x >= 2? No
// Counter Example: Initial State (x = 0)

#assert P |= <> ge2; // Eventually x >= 2?
// Yes. Traces: <a, b>, <a, a>, <b>
```

CSP#: Assertions and Linear Temporal Logic

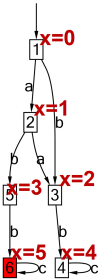
```
var x = 0;
P = [x < 2] a{x = x + 1} -> P
    [][x < 4] b {x = x + 2} -> P
    [][x >= 4] c -> P;
#define ge2 x >= 2;
#define lt2 x < 2;
```




```
#assert P |= [] (lt2 -> X ge2);
// It is always true that if the current
    state is x < 2, it implies the next
    state is x >= 2?
// No. For example, <a>, init state x = 0
    (lt2), next state x = 1(!ge2)

#assert P |= [] (lt2 -> X(Xge2));
// It is always true that if the current
    state is x < 2, then the next next
    state is x >= 2?
// Yes. Traces: <a, a>, <a, b>, <b>
```

CSP#: Assertions and Linear Temporal Logic

```
var x = 0;
P = [x < 2] a{x = x + 1} -> P
    [][x < 4] b {x = x + 2} -> P
    [][x >= 4] c -> P;
#define ge2 x >= 2;
#define lt2 x < 2;
```




```
#assert P |= (lt2 U ge2);
// Is it always x < 2 until x >= 2? Yes

#assert P |= (ge2 R le3);
// x <= 3 until the first position
// where x >= 2? Yes

#assert P |= (ge2 R lt2);
// x < 2 until the first position
// where x >= 2? No
// Counter Examples: <a, b>, <b>
```

CSP# Example: Alternating Bit Protocol

- We are trying to model a simple network protocol that retransmits lost messages between a Sender (A) and Receiver (B).
- A sends a bit to B and message may get lost. Using an internal timer, A should retransmit if there is no ACK when **the time is out**. A should **continue to listen** for the correct ACK if it receives a wrong one.
- B will send ACK if and only if it receives a correct bit. We assume the ACK never gets lost. B should **ignore the wrong bit** received.
- After finishing one bit, A and B should move on to send and ACK the alternating bit.

Note

`ifa (cond){ P } else { Q };` performs the condition checking and the first operation of P or Q together.

CSP# Example: Alternating Bit Protocol

```
channel c 1; // unreliable channel.
channel d 1; // perfect channel.
channel tmr 0; // sender's internal timer.

Sender(alterbit) = (c!alterbit -> Skip [] lost -> Skip); tmr!1 ->
    Wait4Response(alterbit);

Wait4Response(alterbit) = (d?x -> ifa (x == alterbit) {
    tmr!0 -> Sender(1 - alterbit)
} else { Wait4Response(alterbit) }) // Time not out, wait
[] tmr?2 -> Sender(alterbit); // Time is out, retransmit

Timer = tmr?1 -> (tmr?0 -> Timer [] tmr!2 -> Timer);

Receiver(alterbit) = c?x -> ifa (x == alterbit) {
    d!alterbit -> Receiver(1 - alterbit)
} else { Receiver(alterbit) }; // Wait to receive

ABP = (Sender(0) ||| Receiver(0) ||| Timer);
```

CSP# Example: Alternating Bit Protocol

```
1 #assert ABP |= [](c!0 -> <>d?0);
```

Is it always true, that when a Sender process writes 0 into the unreliable channel c, a Receiver process will read the value 0 from the perfect channel d?

- Not valid
- Counter Example:
  $\langle c!0, c?0, d!0, tmr.1, tmr.2, c!0, c?0, tmr.1, tmr.2, c!0, (c!0, c?0, tmr.1, tmr.2, c!0)^* \rangle$

CSP# Example: Shunting Game

Modelling the game board using a 1-D array

```
#define M 7;
#define N 6;
#define o -1;
#define a 1;
#define w 0;
    // col number: 0 1 2 3 4 5 6
var board[N][M] = [o,o,a,a,o,o,o, // 0 row number starting from 0
                   o,o,a,a,o,o,o, // 1
                   a,a,a,w,a,a,a, // 2
                   a,w,a,a,a,w,a, // 3
                   a,w,a,a,a,w,a, // 4
                   o,o,a,w,o,o,o, // 5
                   o,o,a,a,o,o,o];// 6

// Black Position
var r:{0..N-1} = 3; // row
var c:{0..M-1} = 0; // column
```

CSP# Example: Shunting Game

- Modelling the moves

```
Game = [r - 1 >= 0] MoveUp [] [r - 2 >= 0] PushUp
    [] [r + 1 < N] MoveDown [] [r + 2 < N] PushDown
    [] [c - 1 >= 0] MoveLeft [] [c - 2 >= 0] PushLeft
    [] [c + 1 < M] MoveRight [] [c + 2 < M] PushRight;

MoveUp = [board[r - 1][c] == a] go_up{r = r - 1} -> Game;
PushUp = [board[r - 2][c] == a && board[r - 1][c] == w] push_up{
    board[r - 2][c] = w;
    board[r - 1][c] = a;
    r = r - 1;
} -> Game
...
```

- Modelling goal and trouble state.

```
#define goal board[2][2] == w && board[2][3] == w && board[3][2] == w && board
    [3][3] == w;
#assert Game reaches goal;
#define trouble board[0][3] == w;
#assert Game reaches trouble;
#assert Game |= [](trouble -> ! <> goal); // trouble prevents reaching goal
```

CSP# Example: Keyless System

- One of the latest automotive technologies, a push-button keyless system, allows you to start your car's engine without the hassle of key insertion and offers great convenience.
- These systems are designed so it is possible to start the engine without the owner's key-fob and it cannot lock your key fob inside the car because the system will sense it and prevent the user from locking them in.
- However, the keyless system can also suprise you as it may allow you to drive the car without a key-fob, meaning you can drive without physically having the key.
- In this example, we will model such a Keyless System and use assertions to check if one can drive the car without having a key with them.

CSP# Example: Keyless System

Constants and variables

```
#define N 2; // number of owners
#define far 0; // owner is out and far away from the car
#define near 1; // owner is close enough to open / lock the car if he has the keyfob
#define in 2; // owner is in the car
#define off 0; // engine is off
#define on 1; // engine is on
#define unlock 0; // door is unlocked but closed
#define lock 1; // door is locked (and must be closed)
#define open 2; // door is open
#define incar -1; // keyfob is put inside car
#define faralone -2; // keyfob is put outside and far

var owner[N]; // owners' position, initially all users are far away from the car
var engine = off; // engine status, intially off
var door = lock; // door status, initally locked
var key = 0; // key fob position, initially with its first owner
var moving = 0; // car moving status, 0 for stop and 1 for moving
var fuel = 10; // energy costs, say 1 for a short drive and 5 for long driving
```

CSP# Example: Keyless System

- Owner positions

```
car = (||i:{0..N-1} @ (owner_pos(i) || motor(i) || door_op(i) || key_pos(i)));

owner_pos(i) = [owner[i] == far] towards.i{owner[i] = near} -> owner_pos(i)
            [] [owner[i] == near] goaway.i{owner[i] = far} -> owner_pos(i)
            [] [owner[i] == near && door == open && moving == 0]
                getin.i{owner[i] = in} -> owner_pos(i)
            [] [owner[i] == in && door == open && moving == 0]
            goout.i{owner[i] = near} -> owner_pos(i);
```

- Key-fob positions

```
key_pos(i) =
    [key == i && owner[i] == in] putincar.i{key = incar} -> key_pos(i)
 [] [key == i && owner[i] == far] putaway.i{key = faralone} -> key_pos(i)
 [] [(key == faralone && owner[i] == far) || (key == incar && owner[i] == in)]
    getkey.i{key = i} -> key_pos(i);
```

CSP# Example: Keyless System

Door Operation

```
1 door_op(i) =
2     [key == i && owner[i] == near && door == lock && moving == 0]
3     unlockopen.i{door = open} -> door_op(i)
4  [] [owner[i] == near && door == unlock && moving == 0] justopen.i{door = open} ->
      door_op(i)
5  [] [door != open && owner[i] == in] insideopen.i{door = open} -> door_op(i)
6  [] [door == open] close.i{door = unlock} -> door_op(i)
7  [] [door == unlock && owner[i] == in] insidelock.i{door = lock} -> door_op(i)
8  [] [door == unlock && owner[i] == near && key == i] outsidelock.i{door = lock} ->
      door_op(i);
```

CSP# Example: Keyless System

Motor

```
motor(i) =
    [owner[i] == in && (key == i || key == incar) && engine == off && fuel != 0]
    turnon.i{engine = on} -> motor(i)
 [] [engine == on && owner[i] == in && moving == 0] startdrive.i{
        moving = 1;
    } -> motor(i)
 [] [moving == 1 && fuel != 0] shortdrive.i{
        fuel = fuel - 1;
        if (fuel == 0) {engine = off; moving = 0;}
    } -> motor(i)
 [] [moving == 1 && fuel > 5] longdrive.i{
        fuel = fuel - 5;
        if (fuel == 0) { engine = off; moving = 0; }
    } -> motor(i)
 [] [engine == on && moving == 1 && owner[i] == in] stop.i{moving = 0;} -> motor(i)
 [] [fuel == 0 && engine == off] refill{fuel = 10} -> motor(i)
 [] [engine == on && moving == 0 && owner[i] == in] turnoff.i{
        engine = off;
    } -> motor(i);
```

CSP# Example: Keyless System

Reasoning

```
#define keylockinside (key == incar && door == lock && owner[0] != in && owner[1] != in);
#define drivewithoutengineon (moving == 1 && engine == off);
#define drivewithoutkeyholdbyother (moving == 1 && owner[1] == in && owner[0] == far && key == 0);

#assert car deadlockfree;
#assert car reaches keylockinside; // False
#assert car reaches drivewithoutengineon; // False
#assert car reaches drivewithoutkeyholdbyother; // True
```

CSP# Example: Multi-lift System (using C#)

In this example, we are modelling a multiple lift system in a building with multiple floors.

```
#define NoOfFloors 2; // floor-0, floor-1
#define NoOfLifts 2; // lift-0, lift-1
#define NoOfUsers 2; // user-0, user-1

// this array models the external requests; extrequestsUP[i] = 1 denotes there is an upward floor request at floor-i; 0 for no request;
var extrequestsUP[NoOfFloors];
var extrequestsDOWN[NoOfFloors];
// this array models the internal requests;
// intrequests[i][j] = 1 if there is a request for floor-i for lift-j; 0 for no request;
var intrequests[NoOfLifts][NoOfFloors]; // [-1, -1]

// the level of the lift door opens at. -1 means the door is closed
var door = [-1(NoOfLifts)];
```

CSP# Example: Multi-lift System (using C#)

Data Operations to clear internal and external requests when the door of the lift-i is open at each level.

```
door[i] = level;
intrequests[i][level] = 0;
if (direction > 0) {
    extrequestsUP[level] = 0;
} else {
    extrequestsDOWN[level] = 0;
}
```

CSP# Example: Multi-lift System (using C#)

Data Operations (using a function defined in an external C# library) to check whether to continue travelling on the same direction or to change direction.

```
1  public static bool CheckIfToMove(int level, int direction, int i, int NoOfFloors,
       int[] intrequests, int[] extrequestsUP, int[] extrequestsDOWN) {
2      int Counter = level + direction;
3      while (Counter >= 0 && Counter < NoOfFloors) {
4          if (extrequestsUP[Counter] != 0 || extrequestsDOWN[Counter] != 0 ||
       intrequests[i * NoOfFloors + Counter] != 0) {
5              return true;
6          } else {
7              Counter = Counter + direction;
8          }
9      }
10     return false;
11 }
```

CSP# Example: Multi-lift System (using C#) Modelling the Lift

```
1  Lift(i, level, direction) = ifa(intrequests[i][level] != 0 || (direction == 1 &&
       extrequestsUP[level] == 1) || (direction == -1 && extrequestsDOWN[level] == 1)) {
2      opendoor.i.level{ *data operations to clear request* } -> close.i.level{door[i]
       = -1;} -> Lift(i, level direction)
3  } else {
4      checkIfToMove.i.level -> ifa(call(CheckIfToMove, level, direction, i,
5      NoOfFloors, intrequests, extrequestsUP, extrequestsDOWN)) {
6          moving.i.level.direction ->
7          ifa (level + direction == 0 || level + direction == NoOfFloors - 1) {
8              Lift(i, level + direction, -1 * direction)
9          } else {
10             Lift(i, level + direction, direction)
11         }
12     } else {
13         ifa ((level == 0 && direction == 1) ||
14              (level == NoOfFloors - 1 && direction == -1)) {
15             Lift(i, level, direction)
16         } else {
17             changedir.i.level -> Lift(i, level, -1 * direction)
18         }
19     }
20 }
```

CSP# Example: Multi-lift System (using C#) Modelling the Users

```
1  User() = []pos:{0..NoOfFloors - 1}@ (ExternalPush(pos); UserWaiting(pos));
2
3  // The following models the behaviours of the user pushing external buttons
4  ExternalPush(pos) = ifa(pos != 0) {pushdown.pos{extrequestsDOWN[pos] = 1;} -> Skip}
5                      else {pushup.pos{extrequestsUP[pos] = 1;} -> Skip}
6  [] ifa(pos != NoOfFloors - 1) {pushup.pos{extrequestsUP[pos] = 1;} -> Skip}
7      else {pushdown.pos{extrequestsDOWN[pos] = 1;} -> Skip};
8
9  // The following models the behaviours of the user waiting and entering the lift
10 UserWaiting(pos) = []i:{0..NoOfLifts - 1}@([door[i] == pos]enter.i ->
11     ([]y:{0..NoOfFloors - 1}@push.y{intrequests[i][y] = 1;} ->
12         ([door[i] == y]exit.i -> User())));
```

CSP# Example: Multi-lift System (using C#) Questioning the System

```
1 // A Lift System consisting of multiple users and lifts running in parallel
2 LiftSystem() = (||| {NoOfUsers} @ User()) |||
3     (||| x:{0..NoOfLifts - 1} @ Lift(x, 0, 1));
4
5 // If there is an external request at the first floor, it will eventually be served.
6 #define on extrequestsUP[0] == 1;
7 #define off extrequestsUP[0] == 0;
8 #assert LiftSystem() |= [](on -> <>off);
```

CSP# Example: 2-phase Commit Protocol

```
#define N 2; // number of participants
enum {Yes, No, Commit, Abort}; // constants
channel vote 0;
var hasNo = false;

// The following models the coordinator
Coord() = (|||{N}@ request -> Skip);
          (|||{N}@ vote?vo{if (vo == No) {hasNo = true;}} -> Skip);
          decide -> (([hasNo == false] (|||{N}@inform.Commit -> Skip);
          CoordPhaseTwo(Commit))
          [] ([hasNo == true] (|||{N}@inform.Abort -> Skip); CoordPhaseTwo(Abort)));
CoordPhaseTwo(decC) = |||{N}@acknowledge -> Skip;

// The following models a participant
Part() = request -> execute -> (vote!Yes -> PhaseTwo() [] vote!No -> PhaseTwo());
PhaseTwo() = inform.Commit -> complete -> result.Commit -> acknowledge -> Skip
           [] inform.Abort -> undo -> result.Abort -> acknowledge -> Skip;

#alphabet Coord {request, inform.Commit, inform.Abort, acknowledge};
#alphabet Part {request, inform.Commit, inform.Abort, acknowledge};
System = Coord() || (|||{N}@Part()); // Note: request is a common event between
    Coord and Part and has to be synchronised.
```

Table of Contents

Real-Time System (RTS) Module

- The Real-Time System (RTS) module is an extension of the CSP module with operators which captures quantitative **timing requirements.**
- Let P and Q be processes, while d represents a duration of d time units.

```
P = Wait[t] // delay
P = P timeout[t] Q // timeout
P = P interrupt[t] Q // timed interrupt
P = P deadline [t] // deadline
P = P within[t] // within
```

Timed Process Definition: Wait

- A wait process `Wait[t]` delays the system execution for a period of t time units then terminates.
- In the subsequent slides, a set of inference rules using premises and conclusions will be used to define the behaviour of each timed process.
- Recall that each (V, P) is an ordered pair of values and processes.

Definition 1

$$\frac{t \leq d}{(V, Wait[d]) \xrightarrow{t} (V, Wait[d - t])}$$

Definition 2

$$\frac{}{(V, Wait[0]) \xrightarrow{\tau} (V, Skip)}$$

```
1 P = Wait[t]; P
```

- The starting time of process P is delayed by exactly t time units.
- If the amount of time elapsed is less than the specified duration, then the Wait process will be still active.

Timed Process Definition: Timeout

- The process `P timeout[t] Q` passes control to process `Q` if no event has occured in process `P` before `t` time units have elapsed.
- For instance if process `a -> P timeout [t] Q` engages in event `a` before t time units have elapsed since the process is enabled, then the process is transformed to P.
- If event `a` has not occured by time t, the process transforms to Q (by silent tau-transition).

Invisible Events

1. Let τ (tau) be an invisible event. Example: `tau{pv = x}`
2. $\{$ pv $=$ x $\}$ (Event with no name is also an invisible event)

Note: Invisible events are not observable.

Timed Process Definition: Timeout

- The timeout constraint is removed if an event in P is engaged before d time units has elapsed.

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \text{ timeout[d] } Q) \xrightarrow{x} (V', P')}$$

- If P engages in an invisible event τ, the timeout constraint is not removed, but the process becomes `P' timeout[d] Q`

$$\frac{(V, P) \xrightarrow{\tau} (V, P')}{(V, P \text{ timeout[d] } Q) \xrightarrow{\tau} (V', \text{P' timeout[d] } Q)}$$

- If time has not passed d units, the timeout constraint remains.

$$\frac{(V, P) \xrightarrow{t} (V, P'), t \leq d}{(V, P \text{ timeout[d] } Q) \xrightarrow{t} (V, \text{P' timeout[d - t] } Q)}$$

- If no event has occure by timeout, the process transforms to Q by silent-tau transition.

$$\frac{}{(V, P \text{ timeout[0] } Q) \xrightarrow{\tau} (V, Q)}$$

Timed Process Definition: Timed Interrupt

- Process `P interrupt[t] Q` behaves as P until t time unit elapse and then switches to Q
- For example process `(a -> b -> c -> ...) interrupt[t] Q` may engage in events a, b, c, ... as long as t time units has not elapsed.
- Once t time units have elapse, then the process transforms to Q by silent-tau transition.

Definition 1

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \text{ interrupt[d] } Q) \xrightarrow{x} (V', P' \text{ interrupt[d] } Q)}$$

Definition 2

$$\frac{(V, P) \xrightarrow{t} (V', P'), t \leq d}{(V, P \text{ interrupt[d] } Q) \xrightarrow{t} (V', P' \text{ interrupt[d - t] } Q)}$$

Definition 3

$$\frac{}{(V, P \text{ interrupt[0] } Q) \xrightarrow{\tau} (V', Q)}$$

Timed Process Definition: Deadline

Process `P deadline[t]` is constrained to terminate within t time units.

Definition 1

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \text{ deadline[d]}) \xrightarrow{x} (V', P' \text{ deadline[d]})}$$

Definition 2

$$\frac{(V, P) \xrightarrow{t} (V', P'), t \leq d}{(V, P \text{ deadline[d]}) \xrightarrow{t} (V', P' \text{ deadline[d - t]})}$$

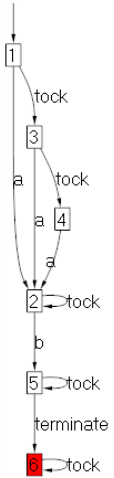
Definition 3

$$\frac{}{(V, P \text{ interrupt[0]}) \rightarrow \mathit{Skip}}$$

Timed Process Definition: Within

- The within operator forces the process to make an observable move within the given time frame.
- For example `P within[t]` says the first event of P must engage within t time units.
- In the example below, event a has to be engaged by latest $t = 2$ which is at **State 4**.

```
1 P = (a -> b -> Skip) within[2];
```



CSP# Example: Fischer's Protocol

Mutual exclusion in Fischer's Protocol is guaranteed by carefully placing bounds on the execution times of the instructions, leading to a protocol which is very simple, and relies heavily on time aspects.

```
#define N 2;
#define Delta 3;
#define Epsilon 4;
#define Idle -1;
var x = Idle;
var counter;

P(i) = ifb(x == Idle) {
    ((update.i{x = i} -> Wait[Epsilon]) within[Delta]);
    ([x == i](cs.i{counter++} -> exit.i{counter--; x=Idle} -> P(i))
    [] [x != i]P(i))
};

FischersProtocol = ||| i:{0..N-1}@P(i);
```

CSP# Example: Railway Crossing System

Control trains passing a critical point (a bridge)

```
#import "PAT.Lib.Queue";
#define N 2;
channel appr 0;
channel go 0;
channel leave 0;
channel stop 0;
var<Queue> queue;

Train(i) = appr!i -> ((stop?i -> StopS(i) within[10]) timeout[10] Cross(i));
Cross(i) = Wait[4]; (leave!i -> Train(i)) within[11];
StopS(i) = go?i -> Start(i);
Start(i) = Wait[10]; (leave!i -> Train(i)) within[10];
Gate = if (queue.Count() ==0) {
    appr?i -> atomic{tau{queue.Enqueue(i)} -> Skip}; Occ
} else {
    (go!queue.First() -> Occ) within[10]
};

Occ = (leave?[i == queue.First()]i -> atomic{tau{queue.Dequeue()} -> Skip}; Gate) []
     (appr?i -> atomic{tau{queue.Enqueue(i)} -> stop!queue.Last() -> Skip}; Occ);
System = (||| x:{0..N-1}@Train(x)) ||| Gate;
```

CSP# Example: Light Control System (Part 1 of 2)

```
var dim : {0..100};
var on = false;
channel button 0;
channel dimmer 0;
channel motion 0;

TurningOn = turnOn{ on = true; dim = 100;} -> Skip;
TurningOff = turnOff{ on = false; dim = 0; } -> Skip;

ButtonPushing = button?1 -> atomic{if (dim > 0) { TurningOff } else { TurningOn }};
DimChange = dimmer?n -> atomic{setdim{dim = n} -> Skip};
ControlledLight = (ButtonPushing [] DimChange); ControlledLight;

// The motion detector
NoUser = move -> motion!1 -> User [] nomove -> Wait[1]; NoUser;
User = nomove -> motion!0 -> NoUser [] move -> Wait[1]; User;
MotionDetector = NoUser;

// Continued on the next slide...
```

CSP# Example: Light Control System (Part 2 of 2)

```
// The room controller
Ready = motion?1 -> button!1 -> On;
Regular = adjust -> dimmer!50 -> Regular;
On = Regular interrupt motion?0 -> OnAgain;
OnAgain = (motion?1 -> On) timeout[20] Off;
Off = button!1 ->> Ready; // Note: ->> is shortcut for atomic
Controller = Ready;

System = MotionDetector ||| ControlledLight ||| Controller;
```

PCSP Module: Probability Processes

- The PCSP module adds **probability processes** to existing process definitions in the CSP# module.
- **Probability processes** are a special kind of process with probabilistic characteristic defined using the keyword `pcase`.
- It is a compositional process made up of probabilistic branches.

```
P = pcase{
    [prob1] : Q1
    [prob2] : Q2
    ...
    default : Qn
}
```

- `prob1` and `prob2` are floating point probability values.
- `default = 1 - prob1 - prob2 - ...`

```
P = pcase{
    weight1 : Q1
    weight2 : Q2
    ...
    weightn : Qn
}
```

- PAT will add up and normalize the weights.
- For example, the probability of P to Q1 is $\frac{weight1}{weight1+weight2+\cdots+weightn}$

Probabilty Processes Assertion

```
1 #assert P reaches cond with prob/pmin/pmax
```

This assertion asks the (min/max/both) probability that the process P() can reach a state at which some given condition is satisifed.

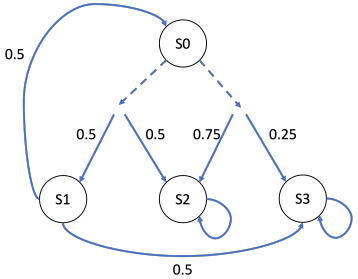
Keyword: `prob`

The keyword `prob` provides both the minimum and maximum probability a process P() can reach a certain state. It provides a range of probabilities.

PCSP Example: Simple pcase

```
var current = 0;
aSystem = State0;

State0 = pcase{
    [0.5] : e05{current = 1} -> State1
    default : e05{current = 2} -> State2
} [] pcase{
    [0.25] : e025{current = 3} -> State3
    default : e075{current = 2} -> State2
};

State1 = pcase{
    [0.5] : e05{current = 0} -> State0
    default : e05{current = 3} -> State3
}

State2 = e -> State2;
State3 = e -> State3;

#define predicate current == 2;
#assert aSystem reaches predicate with pmax; // 0.75
#assert aSystem reaches predicate with pmin; // 0.67
```

PCSP Example: Calculating Max Reachability for Simple pcase



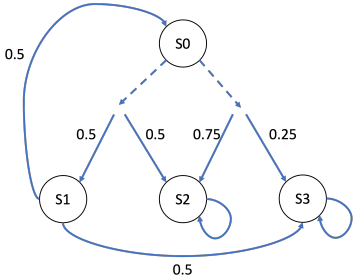
Let x_i be the max reachability from S_i to S_2:

- $x_0 = max(0.5x_1 + 0.5x_2, 0.75x_2 + 0.25x_3)$
- $x_1 = 0.5x_0 + 0.5x_3$
- $x_2 = 1$
- $x_2 = 0$

- x_0, x_1 dependent on other reachability values.
- Initially, assume $x_0, x_1 = 0$.
- When calculating each current iteration, use the previous iteration's x_0 value when calculating x_1. That is why in the table below, we start from iteration 0.
- Stop iterating only when the values are stabilized.

Iteration	x_0	x_1
0	0	0
1	0.75	0
2	0.75	0.375
3	0.75	0.375

PCSP Example: Calculating Min Reachability for Simple pcase



Let x_i be the min reachability from S_i to S_2:

- $x_0 = min(0.5x_1 + 0.5x_2, 0.75x_2 + 0.25x_3)$
- $x_1 = 0.5x_0 + 0.5x_3$
- $x_2 = 1$
- $x_2 = 0$

- x_0, x_1 dependent on other reachability values.
- Initially, assume $x_0, x_1 = 1$.
- Use the previous iteration's x_0 value when calculating x_1.
- Stop iterating only when the values are stabilized.

Iteration	x_0	x_1
0	1	1
1	0.75	0.5
2	0.75	0.375
3	0.6875	0.375
4	0.6875	0.3438
5	0.6719	0.3438
...		
	0.6667	0.3333

PCSP Example: Monty Hall

```
enum{Door1, Door2, Door3};;
var car = -1;
var guess = -1;
var goat = -1;
var final = false;

#define goal guess == car && final;

PlaceCar = []i:{Door1, Door2, Door3}@ placecar.i{car = i} -> Skip;
Goat = []i:{Door1, Door2, Door3}@
    ifb (i != car && i != guess) { hostopen.i{goat = i} -> Skip };

TakeOffer = []i:{Door1, Door2, Door3}@
    ifb (i != guess && i != goat) { changeguess{guess = i; final = true} -> Stop };
NotTakeOffer = keepguess{final = true} -> Stop;

Sys_Take_Offer = PlaceCar; Guest; Goat; TakeOffer;
#assert Sys_Take_Offer reaches goal with prob; // Max Prob = 2/3

Sys_Not_Take_Offer = PlaceCar; Guest; Goat; NotTakeOffer;
#assert Sys_Not_Take_Offer reaches goal with prob; // Max Prob = 1/3
```

PCSP Example: Monty Hall (Explanation)

- What happens if we changed **line 14** to

```
1 if (i != car && i != guess) { hostopen.i{goat = i} -> Skip };
```

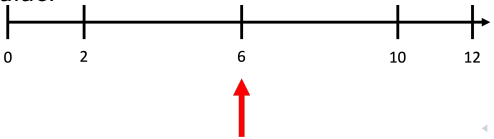
  - goat will remain as -1
  - In the original code, `ifb` was used. The process will wait at the `ifb` block when the condition is not true.
  - However for `if`, the process will terminate if the condition is not true.
  - Alternatively, we can change `ifb` to a guard condition.
- The code in the previous slide **can deadlock**.
  - The `TakeOffer` and `NotTakeOffer` processes have a `STOP` process inside its definition.

PCSP Example: Consensus

```
#define N 2;
#define K 2;
#define range 12; // Range
#define counter_init 6; // Middle
#define left 2; // Left Target
#define right 10; // Right Target
var counter : {0..range} = counter_init; // shared coin
var Pcounter; // record the process number
var coin0counter = N; // number of coints which are 0;
var coin1counter; // number of coints which are 1;
// local variable: 0 - flip, 1 - write, 2 - check, 3 - finished
// processij means this process's coin is i and its local variable is j
process00 = pcase{ [0.5] : process01
                   default : tau {coin0counter--; coin1counter++;} -> process11 };
process01 = [counter > 0] tau {counter--;} -> process02;
process02 = [(counter <= left)] tau {Pcounter++;} -> process03
  [] [(counter >= right)] {Pcounter++; coin0counter--; coin1counter++;} -> process13
  [] [(counter > left) && (counter < right)] process00;
process03 = [Pcounter==N] done -> process03;
process13 = [Pcounter==N] done -> process13;

System = |||{N}@ process00;
```

PCSP Example: Consensus (Explanation)

- In this example, N processes come to a consensus by deciding whether the agreed value should be 0 or 1.
- In `process00`, there is a 50% chance that the process gets coin 0 in the coin flip which goes to `process01`. In the remaining 50% chance, the process gets coin 1 in the coin flip and proceeds to `process11`.
- If a process flips coin 0, the counter's value is reduced by 1.
- If a process flips coin 1, the counter's value is increased by 1.
- If the counter's value is below the left boundary of 2, then the agreed value is 0.
- If the counter's value is below the above boundary of 10, then the agreed value is 1.
- Note that in this system, all N processes are **executing concurrently** without synchronisation. Hence, it is possible that the majority decision on the coin value, may differ from the counter value.



PCSP Example: Consensus (Simplified)

```
#define N 2;
#define K 2;
#define range 12;
#define counter_init 6;
#define left 2;
#define right 6;
var counter: {0..range} = counter_init;
Var Pcounter;

process00 = pcase{
    [0.5] : tau{counter--;} -> process02
     default: tau{counter++;} -> process02
};
process02 = [(counter <= left)] tau{Pcounter++} -> process03;
            [] [(counter >= right)] tau{Pcounter++} -> process13;
            [] [(counter > left) && (counter < right)] process00;
process03 = [Pcounter == N] done -> process03;
Process13 = [Pcounter == N] done -> process13;
System = |||{N}@ process00;
```