Integrated Formal Modeling Techniques and UML

Dr Jin Song Dong
Computer Science Department
National University of Singapore

May 2001

http://www.comp.nus.edu.sg/~dongjs

The 23rd IEEE International Conference on Software Engineering (ICSE 2001, Toronto, Canada) Tutorial Notes (Thanks Drs. Roger Duke and Brendan Mahony for the joint research work and some teaching slides)
Overview

- State-based Formalism: Z/Object-Z
- Event-based Formalism: CSP/Timed-CSP
- Timed Communicating Object Z – TCOZ
- Active Objects and Network Topology
- Case Study: Lift System
- Sensor, Actuator and Control Systems
- Unified Modeling Language (UML)
- Linking TCOZ with UML
- Z family on the Web with their UML pictures

Why Formal Specification?

A formal specification should

- add clarity and understanding by giving a description of the system which is
  - complete
  - unambiguous
  - easily analysed;

- lead to better code that is
  - reliable
  - accurate
  - maintainable
  - reusable
  - verified.
Formal Specification and Software Engineering

(Towards an integrated methodology for software engineering.)

Formal specification

- is not a replacement, but rather an enhancement of existing methodologies;
- can only be effective if integrated within an overall methodology for software engineering.

Implications of using formal specification

- training in the use of notation
- integration with informal methodologies
- translation for client consumption
- emphasis upon abstraction

Some Specification Languages

Process Algebra
Systems modelled as processes partaking in communication:

CSP, CCS, LOTOS

State Oriented
Systems modelled by an underlying state which can undergo change:

VDM, Z, Object-Z

Algebraic
Systems modelled by equations related by axioms (re-writing rules):

ACT 1, CLEAR, OBJ, Larch
The Z Specification Language

- developed originally at Programming Research Group, Oxford University
- based on set theory and predicate logic
- system described by introducing fixed sets and variables and specifying the relationships between them using predicates
- declarative, not procedural
- system state determined by values taken by variables subject to restrictions imposed by state invariant
- operations expressed by relationship between values of variables before, and values after, the operation
- variable declarations and related predicates encapsulated into schemas
- schema calculus facilitates the composition of complex specifications

Types

Z is strongly typed: every expression is given a type.

Any set can be used as a type.

The following are equivalent within set comprehension
\[(x, y): A \times B \quad x : A; \ y : B \quad x, y : A \quad \text{(when } B = A)\]

Notice that
\[\forall S : \mathbb{P} A \bullet \ldots \text{ not } \forall S \subseteq A \bullet \ldots\]

In order to support the use of standard units of measurement, Hayes and Mahony has extended the Z typing system with standard units of measurement, e.g. time quantities are represented by the type
\[T == \mathbb{R}s,\]

which represents real-valued time measured in seconds.
Relations

A relation $R$ from $A$ to $B$, denoted by

$$R : A \leftrightarrow B,$$

is a subset of $A \times B$.

$R$ is the set $\{(c, x), (c, z), (d, x), (d, y), (d, z)\}$

**Notation:** the predicates

$$(c, z) \in R \quad \text{and} \quad c \mapsto z \in R \quad \text{and} \quad c R z$$

are equivalent.

- $\text{dom } R$ is the set $\{a : A \mid \exists b : B \bullet a R b\}$
- $\text{ran } R$ is the set $\{b : B \mid \exists a : A \bullet a R b\}$

Examples

$$\leq : \mathbb{N} \leftrightarrow \mathbb{N}$$

$$\forall x, y : \mathbb{N} \bullet x \leq y \iff \exists k : \mathbb{N} \bullet x + k = y$$

i.e. the relation $\leq$ is the infinite subset

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

of ordered pairs in $\mathbb{N} \times \mathbb{N}$.

$$\text{divides} : \mathbb{N}_1 \leftrightarrow \mathbb{N}$$

$$\forall x : \mathbb{N}_1; y : \mathbb{N} \bullet x \text{ divides } y \iff \exists k : \mathbb{N} \bullet x k = y$$

$$3 \text{ divides } 6 \quad \text{but} \quad \neg (3 \text{ divides } 7)$$
Domain and Range Restriction/Subtraction

Suppose $R : A \leftrightarrow B$ and $S \subseteq A$ and $T \subseteq B$; then

- $S \triangleleft R$ is the set $\{(a, b) : R \mid a \in S\}$
- $R \triangleright T$ is the set $\{(a, b) : R \mid b \in T\}$
- $S \triangleleft R$ is the set $\{(a, b) : R \mid a \notin S\}$
- $R \triangleright T$ is the set $\{(a, b) : R \mid b \notin T\}$

E.g., if

- $\text{has_sibling} : \text{People} \leftrightarrow \text{People}$
- $\text{female} \triangleleft \text{has_sibling}$ is the relation $\text{is_sister_of}$
- $\text{has_sibling} \triangleright \text{female}$ is the relation $\text{has_sister}$
- $\text{female} \triangleleft \text{has_sibling}$ is the relation $\text{is_brother_of}$
- $\text{has_sibling} \triangleright \text{female}$ is the relation $\text{has_brother}$

Functions

A (partial) function $f$ from a set $A$ to a set $B$, denoted by

$$f : A \rightarrow B,$$

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 $(a, b) \in f$. The function $f$ is a total function, denoted

$$f : A \rightarrow B,$$

if and only if dom $f$ is the set $A$.

The predicates

$$(a, b) \in f \quad \text{and} \quad f(a) = b$$

are equivalent.
Examples:

\[
\begin{align*}
\text{root} & : \mathbb{N} \rightarrow \mathbb{N} \\
\text{dom root} & = \{ n : \mathbb{N} \mid \exists m : \mathbb{N} \cdot m^2 = n \} \\
\forall n : \text{dom root} \cdot (\text{root}(n))^2 & = n \\
+ & : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
\forall (n, m) : \mathbb{N} \times \mathbb{N} \cdot +(n, m) & = n + m
\end{align*}
\]

Generic Definitions

\[
\begin{align*}
\lfloor X, Y \rfloor & \\
\text{first} : X \times Y & \rightarrow X \\
\forall x : X; y : Y \cdot \text{first}(x, y) & = x
\end{align*}
\]

Exercise: Generic Purge Function

A generic purge function takes a time period (a timeout of type \( T \)) and a set of time stamped elements of generic type \( X \), and returns a set of updated time stamped elements. For example

\[
ps(2s, \{(1s, a), (3s, b), (7s, c)\}) = \{(1s, b), (5s, c)\}
\]
Solution

\[ X \]
\[ ps : (T \times F(T \times X)) \to F(T \times X) \]
\[ \forall t : T; s : F(T \times X) \bullet \]
\[ ps(t, s) = \{ (t_1, e) : T \times X \mid \exists (t_o, e) : s \bullet t_o - t = t_1 \} \]

or

\[ X \]
\[ \bullet \]
\[ ps(t, s) = \{ (t_o, e) : s \mid t_o > t \bullet (t_o - t, e) \} \]

Any difference between the two?

Sequences

A sequence \( s \) of elements from a set \( A \), denoted

\[ s : \text{seq} \ A, \]

is a function \( s : N \to A \) where \( \text{dom} \ s = 1..n \) for some natural number \( n \). For example,

\[ \langle b, a, c, b \rangle \] denotes the sequence (function){\( 1 \mapsto b, 2 \mapsto a, 3 \mapsto c, 4 \mapsto b \)}

The empty sequence is denoted by \( \langle \rangle \).

The set of all sequences of elements from \( A \) is denoted \( \text{seq} \ A \) and is defined to be

\[ \text{seq} \ A = \{ s : N \to A \mid \exists n : N \bullet \text{dom} \ s = 1..n \} \]

We define \( \text{seq}_1 A \) to be the set of all non-empty sequences, i.e.

\[ \text{seq}_1 A = \text{seq} A - \{ \langle \rangle \} \]

Notice that: \( \langle a, b, a \rangle \neq \langle a, a, b \rangle \neq \langle a, b \rangle \)
Special Functions for Sequences

Concatenation

\[ \langle a, b \rangle \triangleleft \langle b, a, c \rangle = \langle a, b, a, c \rangle \]

Head, Last

\[
\begin{align*}
\text{head, last} : & \text{seq}_1 A \rightarrow A \\
\forall s : & \text{seq}_1 A \Rightarrow \text{head}(s) = s(1) \land \text{last}(s) = s(\#s)
\end{align*}
\]

\[ \text{head} \langle c, b, b \rangle = c \quad \text{last} \langle c, b, b \rangle = b \]

Tail

\[
\begin{align*}
tail : & \text{seq}_1 A \rightarrow \text{seq} A \\
\forall s : & \text{seq}_1 A \Rightarrow \langle \text{head}(s) \rangle \triangleleft \text{tail}(s) = s
\end{align*}
\]

\[ \text{tail} \langle c, b, b \rangle = \langle b, b \rangle \]

Combining Formal Specification with Object-Oriented Design

goes against the conventional view of separating the concerns of functionality and design, but

- adds clarity and leads to simplification of large systems;
- helps with system abstraction and suggests a refinement into object-oriented code.

Object-Z is an extension to Z developed at the University of Queensland. It supports the design of object-oriented design.

Object-Z Basics: Buffer Example

```
Buffer[X]

max : N
items : seq X
\Delta
size : N
\#items = size \land size \leq max

Join
\Delta(items)
i? : X
items' = (i?) \cap items

Leave
\Delta(items)
i! : X
size \neq 0 \land items = items' \setminus \{i\}
```

Exercise: For this Buffer class specify:

(a) an operation count which, given a message, outputs the number of times that message occurs in the buffer;

(b) an operation duplicate which appends to the buffer the message currently at the head of the buffer, provided the buffer is not empty or already full;

(c) an operation titanic whereby a sequence of messages is appended to the buffer except those messages for which there is no room are discarded (the buffer is like a life-boat on the Titanic: people queue to get on, but once the boat is full all the remaining people are left behind);

(d) an operation penguin whereby, like the operation titanic, a sequence of messages is input to the buffer, but this time the messages on the end of the sequence are accepted while those at the front are discarded if there is no room (the messages are acting like penguins, pushing out the messages already in the buffer once the buffer is full).
Solution

```lisp
(count m? : X
count! : N
count! = #(items $ {m?})
)

duplicate

\Delta(items)
\#items \in 1 \ldots (max - 1)
items' = items \leftarrow \langle\text{head items}\rangle

\begin{align*}
titanic & \quad \Delta(items) \\
s? : \text{seq } X & \\
items' = (1 \ldots \text{max}) \leftarrow (\text{items} \leftarrow s?)
\end{align*}

\begin{align*}
penguin & \quad \Delta(items) \\
s? : \text{seq } X & \\
\exists s : \text{seq } X \bullet \\
s \leftarrow \text{items}' = \text{items} \leftarrow s?
\end{align*}

Two Linked Buffers (single thread)

```

\begin{align*}
\text{Two Buffers}[X] & \\
b_1, b_2 : \text{Buffer}[X] & \\
b_1 \neq b_2 & \\
\text{Join} \triangleq b_1.\text{Join} & \\
\text{Leave} \triangleq b_2.\text{Leave} & \\
\text{Transfer} \triangleq b_1.\text{Leave} \parallel b_2.\text{Join} & \\
\end{align*}

\begin{align*}
\text{INIT} & \\
\text{INIT} \land b_2.\text{INIT} & \\
\end{align*}
CSP/Timed CSP

- Hoare’s CSP (Communicating Sequential Processes) an event based notation primarily aimed at describing the sequencing of behaviour within a process and the synchronisation of behaviour (or communication) between processes.

- Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronisation.

Events

A process engages in events; each event is an atomic action. e.g. the events for a vending machine are

- `coin`—insert a coin
- `choc`—extract a chocolate

The set of events that a process can possibly engage in is the alphabet of the process e.g. the alphabet of the vending machine is

```
{coin, choc}
```

Traces

A trace is a finite sequence of events.

A (deterministic) process is specified by the set of traces denoting its possible behaviour. e.g. the traces of the vending machine:

```
( )
(coin)
(coin, choc)
(coin, choc, coin)
...
```

Any execution of the process will be one of these sequences. If \( s \prec t \) is a trace of a process, then so also is \( s \); i.e. the set of traces is prefix closed.
Examples

(1) $\text{STOP}_A$ is the process with alphabet $A$
    that can do nothing.
    
    $\text{traces}(\text{STOP}_A) = \{\langle \rangle\}$

(2) $\text{CLOCK}$ is the process with $\alpha \text{CLOCK} = \{\text{tick}\}$
    which can ‘tick’ at any time.
    
    $\text{traces}(\text{CLOCK}) = \text{tick}^*$

(3) $\text{VM}$ is the process with $\alpha \text{VM} = \{\text{coin}, \text{choc}\}$
    which repeatedly supplies a chocolate after a coin is inserted.
    
    $\text{traces}(\text{VM}) = 
    \{s : \text{seq}\{\text{coin}, \text{choc}\} \mid \exists n : \mathbb{N} \cdot s \leq (\text{coin}, \text{choc})^n\}$

(4) $\text{WALK}$ is a one-dimensional random walk process
    with $\alpha \text{WALK} = \{\text{left}, \text{right}\}$. 
    
    $\text{traces}(\text{WALK}) = (\text{left} \cup \text{right})^*$

Prefix

A process which may participate in event $a$ then act according to process
description $P$ is written

$a \in P(t)$. 

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. The (optional) timing parameter $t$ records
the time, relative to the start of the process, at which the event $a$ occurs and allows
the subsequent behaviour $P$ to depend on its value. For examples:

$\text{VMU} = \text{coin} \rightarrow \text{STOP}$

$\text{SHORTLIFE} = (\text{beat} \rightarrow (\text{beat} \rightarrow \text{STOP})) = \text{beat} \rightarrow \text{beat} \rightarrow \text{STOP}$

$\text{VMS} = \text{coin} \rightarrow \text{choc} \rightarrow \text{STOP}$
Understanding Timed Prefix

Let P be a process which has two free time variables \( t_1 \) and \( t_2 \). A possible execution of the prefix:

\[
\begin{align*}
& a \oplus t_1 \rightarrow b \oplus t_2 \rightarrow P \\
& \downarrow 3 \text{ (time passed)} \\
& a \oplus t_1 \rightarrow b \oplus t_2 \rightarrow P[(t_1 + 3)/t_1] \\
& \downarrow a \text{ (event occur)} \\
& b \oplus t_2 \rightarrow P[3/t_1] \\
& \downarrow 4 \text{ (time passed)} \\
& b \oplus t_2 \rightarrow P[3/t_1][(t_2 + 4)/t_2] \\
& \downarrow b \text{ (event occur)} \\
& P[3/t_1][4/t_2]
\end{align*}
\]

Other CSP/Timed-CSP primitives:

- \( P; Q \) (sequential composition)
- \( P \parallel X \parallel Q \) (synchronous), \( P \parallel || Q \) (asynchronous)
- \( a \rightarrow P \parallel b \rightarrow Q \) (external choice), \( a \rightarrow P \parallel b \rightarrow Q \) (internal choice)
- \( P_1 \lor e \rightarrow P_2 \) (interrupt process)
- \( \text{WAIT } t; P \) (delay), \( a \rightarrow P \triangleright \{t\} \rightarrow Q \) (time-out)
**Sequential Composition**

- The second form of sequencing is process sequencing. A distinguished event $\checkmark$ is used to represent and detect process termination.

- The sequential composition of $P$ and $Q$, written $P; Q$, acts as $P$ until $P$ terminates by communicating $\checkmark$ and then proceeds to act as $Q$.

- The termination signal is hidden from the process environment and therefore occurs as soon as enabled by $P$. The process which may only terminate is written SKIP.

---

**Parallel composition**

The parallel composition of processes $P$ and $Q$, synchronised on event set $X$, is written

$$ P || X || Q. $$

No event from $X$ may occur in $P || X || Q$ unless enabled jointly by both $P$ and $Q$. When events from $X$ do 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 $Q$ separately but not jointly. For example, in the process described by

$$ (a \rightarrow P) || [a] (c \rightarrow a \rightarrow Q) $$

all $a$ events must be synchronisations between the two processes.

In an asynchronous parallel combination

$$ P \parallel Q $$

both components $P$ and $Q$ execute concurrently without any synchronisations.
**Choice**

Diversity of behaviour is introduced through two choice operators.

The external choice operator allows a process a choice of behaviour according to what events are requested by its environment. The process

\[(a \rightarrow P) \sqcap (b \rightarrow Q)\]

begins with both \(a\) and \(b\) enabled. The environment chooses which event actually occurs by requested one or the other first. Subsequent behaviour is determined by the event which actually occurred, \(P\) after \(a\) and \(Q\) after \(b\) respectively.

Internal choice represents variation in behaviour determined by the internal state of the process. The process

\[a \rightarrow P \sqcap b \rightarrow Q\]

may initially enable either \(a\), or \(b\), or both, as it wishes, but must act subsequently according to which event actually occurred. The environment cannot affect internal choice.

---

**Channel**

A channel is a collection of events of the form \(c.n\): the prefix \(c\) is called the **channel name** and the collection of suffixes is called the **values** of the channel.

When an event \(c.n\) occurs it is said that **the value \(n\) is communicated on channel \(c\)**. When the value of a communication on a channel is determined by the environment (external choice) it is called an **input** and when it is determined by the internal state of the process (internal choice) it is called an **output**.

It is convenient to write \(c?n : N \rightarrow P(n)\) to describe behaviour over a range of allowed inputs instead of the longer \(\square n : N \bullet c.n \rightarrow P(n)\). Similarly the notation \(c!n : N \rightarrow P(n)\) is used instead of \(\sqcap n : N \bullet c.n \rightarrow P(n)\) to represent a range of outputs.

**e.g.**

\[COPYBIT = \text{in.0} \rightarrow \text{out.0} \rightarrow COPYBIT \quad \square \text{in.1} \rightarrow \text{out.1} \rightarrow COPYBIT\]

\[\alpha\ COPYBIT = \{\text{in.0, out.0, in.1, out.1}\}\]
**Interrupt**

The interrupt process \( P_1 \lor e \rightarrow P_2 \) behaves as \( P_1 \) until the first occurrence of interrupt event \( e \), then the control passes to \( P_2 \).

**Recursion**

Recursion is used to given finite representations of non-terminating processes. The process expression

\[
\mu \quad P \bullet a?n : N \rightarrow b!f(n) \rightarrow P
\]

describes a process which repeatedly inputs a natural on channel \( a \), calculates some function \( f \) of the input, and then outputs the result on channel \( b \).

**Timeout**

The timeout construct passes control to an exception handler if no event has occurred in the primary process by some deadline.

The process

\[
(a \rightarrow P) \triangleright\{t\} \quad Q
\]

will try to perform \( a \rightarrow P \), but will pass control to \( Q \) if the \( a \) event has not occurred by time \( t \), as measured from the invocation of the process. For example,

\[
MayPrint1 = (receive \rightarrow print \rightarrow STOP) \triangleright\{60\} \ shutdown \rightarrow STOP
\]

\[
MP1(t) = (receive \rightarrow print \rightarrow STOP) \triangleright\{60 - t\} \ shutdown \rightarrow STOP
\]
Exercise: Transmitter

A transmitter which repeatedly send a given message \( x \) until it receives and acknowledgement. Assume that the transmitter is in an environment which is always ready to accept a send message, then it will send the message every 5 time units until an ack message is received. (hint using recursion together with timeout).

Solution

\[
Transmit(x) = send!x \rightarrow ((ack \rightarrow STOP) \triangleright \{5\} Transmit(x))
\]
**Delay**

A process which allows no communications for period $t$ then terminates is written $\text{WAIT } t$. The process

\[
\text{WAIT } t; \ P = \text{STOP } \triangleright \{t\} \ P \\
\sigma \rightarrow P = \sigma \rightarrow \text{WAIT } t; \ P = \sigma \rightarrow (\text{STOP } \triangleright \{t\} \ P)
\]

is used to represent $P$ delayed by time $t$.

---

**State parameters**

In general, the behaviour of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values.

It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal process state.

The approach adopted by CSP is to allow a process definition to be parameterised by state variables. Thus a definition of the form

\[
P_{m:N} \equiv Q(n)
\]

represents a (possibly infinite) family of definitions, one for each possible value of $n$.

There is no inherent notion of process state in CSP, but rather these annotations are a convenient way to provide a finite representation of an infinite family of process descriptions.
Exercise: A generic timed-collection

The generic timed-collection denotes a collection of elements of type $X$ with a time stamp. Operations are allowed to add elements to and delete elements from the collection. When deleting an element from the collection, the oldest element should be removed and output to the environment. The collection has the following timing properties. Firstly, that it updates the internal state during a add or delete operation. Secondly, each element of the collection becomes stale if it is not passed on within $t_o$ time units of being added to the collection. Stale elements should never be passed on, but are instead purged from the collection upon becoming stale.

Solution

$\textsf{TimedCollection} \triangleq TC_\mathcal{G}.$

$TC_\mathcal{G} \triangleq \text{left}\textstyle{\forall} e : X \rightarrow TC_{\{\{t_a,c\}\}}$

$TC_{\{\{t_a,c\}\}} \cup s \triangleq$

$(\text{left}\textstyle{\forall} e : X \rightarrow TC_{ps(t_i, \{\{t_a,c\}\}\cup\{\{t_o,c\}\}} \circ$

$(\text{right}\textstyle{\forall} a @t_i \rightarrow TC_{ps(t_i,s)} \circ \{t\} TC_{ps(t,s)}$

where $(t, a) = \text{find\_oldest}(\{\{t, a\}\} \cup s)$.

\begin{align*}
\text{find\_oldest} & : \mathcal{P}_1(T \times X) \rightarrow (T \times X) \\
\forall s : \mathcal{P}_1(T \times X) & \bullet \\
\exists (t, e) : s \bullet t = \min(\text{dom } s) \\
\text{find\_oldest}(s) & = (t, e)
\end{align*}
**Object-Z and Timed CSP**

- **Object-Z**
  - ✓ an excellent tool for modeling data states
  - × but difficult for modelling real-time concurrent systems
- **Timed CSP**
  - ✓ Good for specifying the timed process and communication
  - × Like CSP, cumbersome to capture the data state of a complex system
- **Timed Communicating Object Z**: a blending of Object-Z and Timed CSP

**Related Work**

* Z/OZ with CSP: Fischer, Smith, Derick, Suhl, Bolton, Davies, Woodcock ...
* Z with CCS: Galloway, Stoddart, Taguchi, Araki ...

---

**Timed Communicating Object Z (TCOZ)**

```
TimedBuffer[X]

init

\[ \text{items} : \text{seq } X \]
\[ \text{left, right : chan} \]

\[ \text{Add} \]
\[ \Delta(\text{items}) \]
\[ \text{i? : X} \]
\[ \text{items'} = (\text{i?}) \wedge \text{items} \]

\[ \text{Remove} \]
\[ \Delta(\text{items}) \]
\[ \text{items'} = \text{items} \bowtie \{\text{last(\text{items})}\} \]

\[ \text{Join} \equiv [i : X] \bullet \text{left}?i \rightarrow \text{Add}; \text{DEADLINE i} \]

\[ \text{Leave} \equiv [\text{items} \neq \langle \rangle] \bullet \text{right!last(\text{items})} \rightarrow \text{Remove}; \text{DEADLINE i} \]

\[ \text{MAIN} \equiv \mu Q \bullet (\text{Join} \parallel \text{Leave}); Q \]
```
Abstract syntax

\[ ZS, ZE \] [Z schemas and expressions]

\[ TZE ::= \] [TCOZ constructor expressions]
  \[ \text{ref} \langle NAME \rangle \mid \text{STOP} \mid \text{SKIP} \mid \text{WAIT} \langle ZE \rangle \mid (\_ \cdot \_\langle ZS \times TZE \rangle) \mid \]
  \[ (\_ \to \_\langle \Sigma \times TZE \rangle) \mid (\_\cdot \_\langle \Sigma \times ZE \times TZE \rangle) \mid \]
  \[ (\_ \square \_\langle TZE \times TZE \rangle) \mid \ldots \mid (\mu \_ \cdot \_\langle NAME \times TZE \rangle) \]

\[ TZB \equiv \{ C : NAME \to TZE \mid \]
  \[ \forall tze : \text{ran} C; \; N : NAME \bullet \]
  \[ N \in \text{sig} tze \Rightarrow N \in \text{dom} C \} \]

\[ TZC_p \equiv [\text{init} : ZS; \; C : TZB] \] [passive classes]

\[ TZC_a \equiv \]
  \[ [\text{init} : ZS; \; C : TZB; \; \text{main} : TZE \mid \text{sig main} \subseteq \text{dom} TZB] \]

TCOZ Semantics

The support of timing primitives in TCOZ is made possible through the adoption of Reed’s timed-failures semantics for Timed CSP. The timed-failures semantics models CSP processes in terms of timed event-traces and timed event-failures. This semantic model allows CSP to be extended with time related primitives such as delays, timeouts, and clock-interrupts. In order to support objects with encapsulated state this model is extended to include an initial state and state update events. Object-Z operations are modelled as terminating sequences of timed state-update events.

Exercise: A generic timed-collection in TCOZ

The generic timed-collection denotes a collection of elements of type $X$ with a time stamp. Operations are allowed to add elements to and delete elements from the collection. When deleting an element from the collection, the oldest element should be removed and output to the environment. The collection has the following timing properties. ...

Solution

\[
\text{TimedCollection}[X]\]

\[
\text{INIT} \quad tc = \emptyset
\]

\[
\text{Add}_0 \quad \Delta(tc) \quad e? : X; t_i : T \\
{} \quad tc' = ps(t_i, tc) \cup \{(t_o, e?)\}
\]

\[
\text{Delete}_0 \quad \Delta(tc) \quad t_i : T \\
{} \quad tc' = ps(t_i, tc \setminus \{(t, \text{oldest})\})
\]

\[
\text{Add} \equiv [e : X; t_i : T] \bullet \text{left?e @t}_i \rightarrow \text{Add}_0 \\
\text{Delete} \equiv [t_i : T] \bullet \text{right!oldest @t}_i \rightarrow \text{Delete}_0 \\
\text{MAIN} \equiv \mu T \bullet [tc = \emptyset] \bullet \text{Add;} T \quad \square \\
{} \quad [tc \neq \emptyset] \bullet ((\text{Add} \circ \text{Delete}) \triangleright\{t\} tc := ps(t, tc); T
\]
The Notion of Active Object

- Active objects have their own thread of control.
- Passive objects are controlled by other objects in a system.
- A class for defining active objects is called an *active class*.
- A class for defining passive objects is called a *passive class*.
- In TCOZ, MAIN, a non-terminating process definition, distinguishes the active and the passive classes.

Inheritance between active/passive classes

- When a new active class is derived from an existing active class, the MAIN process must always be redefined explicitly.
- A new active class can be derived from an existing passive class, in this case, a MAIN process definition needs to be added.
- A new passive class can also be derived from an existing active class, in this case, the MAIN process of the existing class is implicitly removed.
- A new passive class can be derived from an existing passive class following the same rules as the standard Object-Z.
Composition and interaction of active objects

\[
\begin{align*}
A & \quad B \\
\ \ \ \ \ \ v : T; \ldots & \quad \quad a : A \\
\ \ \ \ \ \ c : \text{chan}; \ldots & \\
\text{Op}A_1 \cong \ldots & \\
\ldots & \\
\text{MAIN} \cong \ldots & \\
\text{Op}B_1 \cong a.\text{Op}A_1 & \\
\ldots & \\
\end{align*}
\]

Identifying the object name with its \text{MAIN} process, e.g. if \text{ob}_1 and \text{ob}_2 are active object components, then \text{ob}_1 || \text{ob}_2 means \text{ob}_1.\text{MAIN} || \text{ob}_2.\text{MAIN}.

Two Communicating Timed Buffers

\[
\begin{align*}
\text{Timed-Buffer} & \quad \text{Timed-Buffer} \\
l & \quad l : \text{TimedBuffer}[X][\text{middle}/\text{right}] \\
& \quad r : \text{TimedBuffer}[X][\text{middle}/\text{left}] \\
\text{MAIN} & \quad \cong (l || \text{middle}) \conc r \conc \text{middle}
\end{align*}
\]
Complex network topologies

\[(A[bc'/bc] \parallel [ab, ac]) (B[ac'/ac] \parallel [bc]) C[ab'/ab]) \setminus ab, ac, bc) ](ab, ac, bc/ab', ac', bc')\]

\[\{v_1, v_2, v_3\... v_1 \leftrightarrow v_2, v_2 \leftrightarrow v_3, v_3 \leftrightarrow v_1, \...\} (A, B, C)\]

\[\{ A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A\} \quad \text{or} \quad \{ A \leftrightarrow B, B \leftrightarrow C, C \leftrightarrow A\}\]

The Lift Case Study

- Multi-floors with multi-elevators
- Non-trivial
- Commonly used example
- Both CSP and Object-Z have been applied (but no real-time issues)
Detailed model can be found at:
http://www.comp.nus.edu.sg/~dongjs/papers/tse00.ps
Lift door control

\[ \text{Door} \]

\[
\text{open, conf, close, servo, sensor : chan}
\]

\[ \text{OpenDoor, CloseDoor} \equiv \ldots \]

\[ \text{CycleDoor} \equiv \text{OpenDoor}; \text{conf} \rightarrow \]

\[
(\mu \text{CD} \cdot \text{Wait} \; t; \text{CloseDoor} \lor \{\text{sensor?(self, Interrupt)}\} \text{OpenDoor}; \text{conf} \rightarrow \text{CD})
\]

\[ \text{MAIN} \equiv \mu \text{D} \cdot \text{open} \rightarrow \text{CycleDoor}; \text{close} \rightarrow \text{D} \]

Moving the lift

\[ \text{Shaft} \]

\[
\text{move, arrive : chan}
\]

\[ \text{MAIN} \equiv \mu \text{S} \cdot \text{move}?n \rightarrow \text{Wait}[n] \cdot \text{I} + \text{delay}; \text{arrive} \rightarrow \text{S} \]
\[ \text{Internal}_Q \]
\[
\begin{aligned}
\text{panel} & : \text{seq Button} \\
\text{int}_\text{request}, \text{int}_\text{sched}, \text{int}_\text{serv} & : \text{chan}
\end{aligned}
\]
\[
\text{NextUp}, \text{NextDown}, \text{MAIN} \triangleq \ldots
\]

\[ \text{Lift}_\text{Control} \]
\[
\begin{aligned}
\beta & : \mathbb{N} \\
\text{md} & : \text{MoveDirection} \\
\text{move}, \text{arrive} & : \text{chan} [\text{shaft}] \\
\ldots
\end{aligned}
\]
\[
\text{MAIN} \triangleq \mu \text{LC} \\
\ldots \rightarrow \text{Internal}; \text{LC} \square \\
\ldots \rightarrow \text{External}; \text{LC}
\]

\[ \text{Lift} \]
\[
\begin{aligned}
\text{iq} & : \text{Internal}_Q \\
\text{lc} & : \text{Lift}_\text{Control} \\
\text{s} & : \text{Shaft} \\
\text{d} & : \text{Door}
\end{aligned}
\]
\[
\text{MAIN} \triangleq \left\langle \begin{array}{l}
\text{lc} \leftarrow \text{move, arrive} \\
\text{lc} \leftarrow \text{open, close, or stay open} \\
\text{lc} \leftarrow \text{int}_\text{sched, int}_\text{serv, or int}_\text{serv}
\end{array} \right\rangle
\]

\[ \text{Lifts} \]
\[
\begin{aligned}
\text{lifts} & : \mathcal{P} \text{Lift} \\
\text{MAIN} \triangleq \left\langle \begin{array}{l}
l \text{ : lifts}
\end{array} \right\rangle
\]
The Lift System

```
requests : seq(N \times MoveDirection)
enter, select, visit, service : chan

Join ________________ Remove ________________
... ________________  ... ________________

Dispatch \triangleq [... \bullet select!item \to Remove

CheckServ \triangleq
  [item : N \times MoveDirection] \bullet visit?item \to ...

MAIN \triangleq \mu C \bullet (Join \Box Dispatch \Box CheckServ); C
```

```
LiftSystem

fs : Floors
ls : Lifts
contr : Controller

MAIN \triangleq ((fs \rightarrow contra \rightarrow select \rightarrow ls \rightarrow service \rightarrow fs))
Sensors and Actuators — Control Systems

- CSP channel mechanism is discrete
- CSP channel mechanism is synchronous

Example: Digital Temperature Display

No Wonder its So Hot

37.6 C

On

Off
Figure 3: The office communication scenario.

\[\text{temp} : \mathbb{R} \to \mathbb{C}_{\text{sensor}}, \quad \Rightarrow \quad \text{temp} : \mathbb{R} \to \mathbb{R}^\circ.\]

Internally, temp takes the syntactic role of a CSP channel. Whenever a value \(v\) is communicated on the internal channel at a time \(t\), \(\text{temp}(t) = v\).

\[\text{screen} : \text{Display actuator},\]

where

\[\text{Display} ::= \text{Temp}(\langle N \ast 0.5^\circ \rangle) \mid \text{nil}.\]

The internal role is that of the local state variable.
Asynchronous active object

Synchronous active objects

- have discrete interfaces, synchronous channels;
- are highly dependent.

Asynchronous active objects

- have analog interfaces, asynchronous sensor/actuators;
- are highly independent;
- can be further classified into periodic and non-periodic objects.
Exercise: a calendar clock

A typical periodic object: a calendar clock ticks every second ...

$\textit{CalendarTime} \equiv N_y r \times N_{mn} \times N_{dy} \times N_{hr} \times N_{min} \times N_s.$

$\textit{Convert} : N_s \rightarrow \textit{CalendarTime}$

... [detail of function omitted]

Solution

$\textit{Clock}$

per $\equiv$ 1s

gain $\equiv$ 50 ms

\[
\begin{array}{c}
\text{total} : N_s \\
\Delta \\
\text{display} : \textit{CalendarTime actuator} \\
\text{display} = \text{Convert}(\text{total})
\end{array}
\]

\[
\begin{array}{c}
\text{Inc} \\
\Delta(\text{total}) \\
\text{total}' = \text{total} + 1
\end{array}
\]

\[
\text{Main} \triangleq \mu C \cdot \text{Inc} ; \ \text{Deadline} \cdot \text{gain} ; \ \text{WaitUntil} \per ; \ C
\]
**UML**

- UML stands for Unified (?) Modeling Language
- The UML combines/collcts Data Modeling concepts (Entity Relationship Diagrams), Business Modeling (work flow), Object Modeling, and Component Modeling
- The UML is the OMG standard language for visualising, specifying, constructing, and documenting the artifacts of a software-intensive system
- UML consists of use case, class, statechart, collaboration diagrams ...

---

**Use Case Diagram**

- Use case is a pattern of behavior the system exhibits. Each use case is a sequence of related transactions performed by an actor and the system in a dialogue. Actors are examined to determine their needs. Use case diagrams are created to visualise the relationships between actors and use cases
Class Diagram

- A class diagram shows the existence of classes and their relationships in the logical view of a system. It consists of classes and their structure and behavior, association, aggregation, dependency, and inheritance relationships, multiplicity and navigation indicators, and role names.

Collaboration Diagram – dynamic behavior, message-oriented

- A collaboration diagram displays object interactions organised around objects and their links to one another.
Statechart Diagram – dynamic behavior, event-oriented

- A statechart diagram shows the life history of a given class, the events that cause a transition from one state to another, and the actions that result from a state change.

Shortcomings of UML

- There is no unified formal semantics for all those diagrams. There are a few approaches to formalize a subset of UML, e.g., (Evans and Clark, 1998, Kim and Carrington, 1999) concentrated on class diagram semantics. Therefore, the consistency between diagrams is problematic; and

- There are limited capabilities for precisely modeling timed concurrency. For example, (in a new feature that has been added to the UML 1.3) synchronisation between concurrent substates of a single statechart diagram can be captured using a synch state link. However there is no facility to precisely model synchronous interactions between states in two different statechart diagrams.
Linking TCOZ and UML

- Syntactically, UML/OCL (Object Constraint Language) is extended with TCOZ communication interface types — chan, sensor and actuator. Upon that, TCOZ sub-expressions can be used (in the same role as OCL) in the statechart diagrams and collaboration diagrams.

- Semantically, UML class diagrams are identified with the signatures of the TCOZ classes. The states of the UML statechart are identified with the TCOZ processes (operations) and the state transition links are identified with TCOZ events/guards.

- Effectively, UML diagrams can be seen as the viewpoint visual projections from a formal TCOZ model.

Combination Process of TCOZ and UML

1. Firstly, the UML use-case models (user-case and collaboration diagrams) are used to analyse system requirements so that main classes and operations will be identified (e.g. classification of the boundary and control classes). Communication links of the collaboration diagrams guide the design of communication interfaces of the TCOZ model (synchronisation — channel, synchronisation — sensor/actuator).

2. Then, the UML class diagrams are used to capture the static structure of the system, in which class/object relationships can be captured.

3. Based on UML class diagrams, detailed TCOZ formal models are constructed in a bottom-up style. The states, timing and concurrent interactions of the system objects are captured precisely in the TCOZ models.

4. Finally, UML state diagrams are used to visualize the behaviors (process states and events) of essential components of the system, which are closely associated with the behavior parts of the TCOZ model.
Class

Synchronisation

A

\[ c : \text{chan} \]

\[ \text{Main} \equiv \ldots c! \ldots \]

B

\[ c : \text{chan} \]

\[ \text{Main} \equiv \ldots c? \ldots \]

AB

\[ a : A \]
\[ b : B \]

\[ \text{Main} \equiv \ldots a \rightarrow b \ldots \]
**Asynchronisation**

\[ A \]
\[ \text{\vspace{0.5cm} \begin{itemize} \item N actuator \end{itemize}} \]
\[ \text{\vspace{0.5cm} ......} \]
\[ B \]
\[ \text{\vspace{0.5cm} \begin{itemize} \item N sensor \end{itemize}} \]
\[ \text{\vspace{0.5cm} ......} \]
\[ AB \]
\[ \text{\begin{itemize} \item a : A \item b : B \end{itemize}} \]
\[ \text{Main} \Rightarrow \ldots \text{a} \leftrightarrow \text{b.} \]

---

**Dynamic Behavior**

\[ P_1; e \rightarrow P_2 \]

\[ P_1; ([\text{guard1}}) \bullet P_2 \sqcup [\text{guard2}}) \bullet P_3) \]

\[ P_1 \parallel P_2 \parallel P_3 \]
**Light Control System (LCS)**

In most existing light control systems, all lights are controlled manually. Electrical energy is wasted by lighting rooms that are not occupied and by not adjusting light levels relative to need and daylight. LCS is an intelligent control system. It can detect the occupation of the building, then turn on or turn off the lights automatically. It is able to tune illumination in the building according to the outside light level. It gains input from sensors and actuators.

\[ \text{Illumination} == 1..10000 \text{ lux} \]
\[ \text{Percent} == \{0\} \cup 10..100 \]

**MotionDetector**

\[
\begin{align*}
\text{motion} & : \text{chan} \\
\text{md} : (\text{Move} | \text{NoMove}) & : \text{sensor} \\
\text{NoUser} & \triangleq \text{md}?\text{Move} \rightarrow \text{motion!1} \rightarrow \text{User} \quad \Box \\
\text{md}?\text{NoMove} & \rightarrow \text{Wait 1 s}; \text{NoUser} \\
\text{User} & \triangleq \text{md}?\text{NoMove} \rightarrow \text{motion!0} \rightarrow \text{NoUser} \quad \Box \\
\text{md}?\text{Move} & \rightarrow \text{Wait 1 s}; \text{User} \\
\text{MAIN} & \triangleq \text{NoUser}
\end{align*}
\]
\[ \text{Light} \]

\[
\begin{align*}
\text{dim} : \text{Percent} & \quad \text{actuator} \\
\text{on} : \mathbb{B} & \quad \text{[dim value]} \\
\end{align*}
\]

TurningOn \( \triangleq \) dim := 100; on := true
TurningOff \( \triangleq \) dim := 0; on := false

\[ \text{ControlledLight} \]

\[
\begin{align*}
\text{Light} & \\
\text{button}, \text{dimmer} : \text{chan} & \quad \text{[control channels]} \\
\end{align*}
\]

ButtonPushing \( \triangleq \) button?1 \( \rightarrow \)
\[
([\text{dim} > 0] \bullet \text{TurningOff} \; \square [\text{dim} = 0] \bullet \text{TurningOn})
\]

DimChange \( \triangleq \) \( [n : \text{Percent}] \bullet \text{dimmer}?n \rightarrow \)
\[
([\text{on}] \bullet \text{dim} := n \; \square [\neg \text{on}] \bullet \text{SKIP})
\]

MAIN \( \triangleq \mu N \bullet (\text{ButtonPushing} \; \square \text{DimChange}) ; N \)
satisfy : Percent ↔ Illumination

RoomController

\[
\begin{align*}
\text{dimmer, motion : chan} & \quad \text{Adjust} \\
\text{od sensor : Illumination sensor} & \quad \text{dim! : Percent on dimmer} \\
\text{absent : T} & \quad \text{dim! satisfy olight}
\end{align*}
\]

\[
\begin{align*}
\text{Ready} & \equiv \text{motion}\!?:1 \rightarrow \text{On} \\
\text{Regular} & \equiv \mu R \cdot [n : \text{Illumination}] \cdot \\
& \quad \text{od sensor}\!?:n \rightarrow \text{olight} := n; \text{Adjust}; \text{dimmer}\!!:\text{dim} \rightarrow R \\
\text{On} & \equiv \text{Regular} \lor \text{motion}\!?:0 \rightarrow \text{OnAgain} \\
\text{OnAgain} & \equiv (\text{motion}\!?:1 \rightarrow \text{On}) \triangleright \{\text{absent}\} \text{ Off} \\
\text{Off} & \equiv \text{dimmer}\!?:0 \rightarrow \text{Ready} \\
\text{MAIN} & \equiv \text{Off}
\end{align*}
\]

LCS

\[
\begin{align*}
m : \text{MotionDetector} \\
l : \text{ControlledLight} \\
r : \text{RoomController}
\end{align*}
\]

\[
\text{MAIN} \equiv \big\| (m \overset{\text{motion}}{\longrightarrow} r \overset{\text{dimmer}}{\longrightarrow} l)\big\|
\]

1: motion  \quad 2: dimmer

- MotionDetector  \quad - RoomController  \quad - ControlledLight

86

87
Z Family on the Web with their UML Photos

- Use eXtensible Markup Language (XML) to develop web environment for Z family languages
  - share design models
  - hyperlinks among models
  - advance browsing facilities

http://nt-appn.comp.nus.edu.sg/fm/zml/

- Develop techniques for projecting (object-oriented) Z models to UML diagrams, based on XML Metadata Interchange (XMI).
- Use Object-Z to specify and design the essential functionalities of the ZML environment
Formal Object Design of ZML

**UMLClass**
- **name**: String
- **attris**: String → DType
- **ops**: D String

**UMLDiagram**
- **classes**: D UMLClass
- **inh, agg**: UMLClass → UMLClass
- \( \forall (inh \cup agg) \cup ran(inh \cup agg) \subseteq classes \)
- \( \forall h : classes \bullet (\bar{a}, h) \notin inh \)

**project**: D Classdef → UMLDiagram

\( \forall (oz, uml) : project \bullet \)
- \( \{ c : oz \bullet c.name \} = \{ c : uml.classes \bullet c.name \} \bullet \)
- \( \forall c_1, c_2 : oz \bullet \)
  - \( \exists c' : uml.classes \bullet \)
    - \( c'.name = c_1.name \)
    - \( c'.attris = \{ cls : oz \bullet cls.name \} \cup c_1.state.dexport \)
    - \( c'.ops = \{ o : Opdef \bullet o \in c_1.ops \bullet o.name \} \)
  - \( c_2.name \in \{ t : ran c_1.state.dexport \bullet t.name \} \Rightarrow \)
  - \( \forall (c'_1, c'_2) : uml.agg \bullet c'_1.name = c_1.name \land c'_2.name = c_2.name \)
  - \( c_2.name \in \{ inh : dom c_1.inherit \bullet inh.name \} \Rightarrow \)
  - \( \exists (c'_1, c'_2) : uml.inh \bullet c'_1.name = c_1.name \land c'_2.name = c_2.name \)
Basic Implementation Ideas

- ZML: Define a customized XML for Z family languages for web-browsing purpose
- UML tool: Rational Rose 2000 supports XMI import/export according to UML DTD
- Translation rules are applied using XSLT techniques to automatically translate Object-Z/TCOZ model(XML) to UML diagrams(XMI) and vice versa

Syntax definition

```xml
<ElementType name="op" content="eltOnly" order="seq">
  <element type="name" minOccurs="1" maxOccurs="1"/>
  <element type="delta" minOccurs="0" maxOccurs="1"/>
  <element type="decl" minOccurs="0" maxOccurs="*"/>
  <element type="predicate" minOccurs="0" maxOccurs="*"/>
  ...
</ElementType>

<ElementType name="classdef" content="eltOnly">
  <element type="state" minOccurs="1" maxOccurs="1"/>
  <element type="init" minOccurs="0" maxOccurs="1"/>
  <element type="op" minOccurs="0" maxOccurs="*"/>
  ...
</ElementType>
```
**XSL Transformation**

```xml
<xsl:template match="classdef[@layout='simp1'] classdef[@layout='gen']">
  <html>
    ...
    <a><xsl:attribute name="name"><xsl:value-of select="name"/></xsl:attribute></a>
    ...
    <xsl:apply-templates select="state"/>
    <xsl:apply-templates select="init"/>
    <xsl:apply-templates select="op"/>
    ...
  </html>
</xsl:template>
```

**Light example**

```xml
<classdef layout="simp1" align="left">
  <name>Light</name>
  <state>
    <decl>
      <name>dim</name>
      <dttypedesc><type>Percent</type><type>&amp;actuator;</type></dttypedesc></decl>
    <decl>
      <name>on</name>
      <dttypedesc><type>&amp;bool;</type></dttypedesc></decl>
  </state>
  <op layout="calc">
    <name>TurningOn</name>
    <predicate>dim := 100; on := true</predicate>
    ...
  </op>
</classdef>
```
It's the time to conclude
Conclusion and Further Work

- State-based (Object-Z), Event-based (Timed CSP), Graph-based (UML)
- TCOZ
  - combines the modelling powers from Object-Z and Timed CSP
  - distinguishes the notion of active and passive objects
- Further research
  - applications to the specification of
    - software architectures
    - parallel distributed systems
  - tools support
  - Hoare and He's UTP to TCOZ semantics
  - TCOZ refinement rules

TCOZ papers


Most online versions can be found at: http://www.comp.nus.edu.sg/~dongjs
Other integrated approaches (partial collection)


