Lecture 2: Outline

- System modelling and design
- The Event-B modelling and verification framework
- The Event-B model structure
- The mathematical foundations of Event-B
- Examples

System Modelling vs System Design

- It is about producing designs for systems that behave consistently with respect to the requirements for the system
- As a programmer, you generally move from your understanding of the system to be developed directly to an implementation
- Question: why does the resultant system behaves according to the requirements? You would probably find it very difficult to answer that question
- In this course we are tackling the question from the other direction: we start with the required behaviour (model) and proceed from there

Engineering, not blacksmithing

- What we are aiming for is engineering. Engineers should be able to explain any system they claim to have designed
- Blacksmiths bashes a piece of metal until it looks like what is desired. Many programmers do the same
- Most of the common programming "methods" do not have any concept of measurements, so how to show why your program does what you claim (believe) it does?
- Rigorous inspection/verification of system designs (models) or prototypes – one way to answer that question

Event-B

In this course, we will be using a modelling method named Event-B. Event-B models represent a system by specifying

- a state: consisting of a set of variables, who values collectively define the system state;
- events: describing changes that can happen to the state.

In turn, the model events consist of:

- parameters: values that can be used to control events;
- guards: boolean conditions on the state and the parameters that define the cases for which an event is enabled (active);
- actions: the change of state that will occur when the event is executed.

Events and Requirements

A careful consideration of the above description of an event will show that events are perfect for formalising a requirement:

- what is required to happen;
- the conditions under which it should happen;
- any parameters that affect the requirements.

Formal development by refinement

Event-B also supports system gradual system development by refinement, when the system details are revealed step-by-step. So it combines

- Abstract modelling
  - Helps to cope with complexity;
  - Focus on stating requirements and assumptions;
  - Allows us to spot requirements ambiguities and contradictions.

- Refinement
  - Elaboration on abstract models;
  - Structuring of requirements;
  - (Automated) proof of adherence to the abstract model.

Modelling in Event-B

- The dynamic system behaviour is described in terms of guarded commands (events):
  - Stimulus $\rightarrow$ response.

- General form of an event:

WHERE *guard* **THEN** *action* **END**

where

  - *guard* is a state predicate defining when an event is enabled;
  - *action* is (possibly non-deterministic) update of state variables.

- If the event guard is missing (i.e., always true), an event can be simply defined as:

BEGIN *action* **END**

Modelling operations in Event-B (cont.)

- General form of a parameterised event:

  ev = **ANY** *pars* **WHERE** *guard* **THEN** *action* **END**

  where
  - *pars* are event parameters and/or local variables;
  - *guard* is condition on the machine state and event parameters/local variables.
- If several events are enabled at the same time (i.e., their conditions are "overlapping"), any of them can be chosen for execution.
  A simple case of nondeterminism, when we have no control or information about which happens first

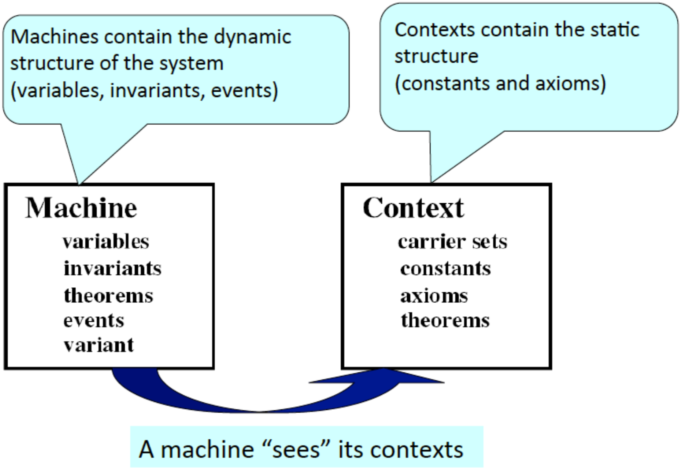
Modelling in Event-B

- Overall system behaviour: a (potentially) infinite loop of system events:

```
forever do
    Event1 or
    Event2 or
    Event3 or ...
end
```

- Model invariant defines a set of allowed (safe) states;
  - Each event should preserve the invariant;
  - We should verify this by proofs.

A system model in Event-B



Structure of a model context component

CONTEXT	model context name
SETS	local types
CONSTANTS	local constants
AXIOMS	conditions/constraints on sets and constants

- Context describes the static data structures to be used in the model machine component
- Axioms essentially contain definitions of the introduced local types (sets) and constants
- Both sets and constants can be abstract (under-specified), allowing for many concrete implementations
- In that case, those sets and constants can be considered as model parameters

Structure of a model machine component

MACHINE	machine name
SEES	model context name
VARIABLES	machine variables
INVARIANT	invariant properties
INITIALISATION	variable initialisation (assignment)
EVENTS	machine events, guards and actions

- Machine describes the system dynamics as all possible state (variable) changes
- The machine invariant contains the properties to be maintained (preserved) in all system states
- Variable typing properties are trivial system invariants
- Machine events contain system guarded (and possibly parameterised) reactions in the form of (parallel) state assignments

A trivial example: Booking system

Simple requirements

1. The booking system allows the customers to book tickets to the scheduled events
2. A customer may book a number of tickets
3. A customer may cancel his/her tickets
4. The booking system should keep track of the currently available tickets
5. The overall number of tickets cannot exceed the pre-defined number (the capacity of a venue)
6. A customer may be restricted by the maximal number of tickets he or she may book

A trivial example: Booking system

MACHINE
  Booking
VARIABLES
  seats
INVARIANT
  $seats \in \mathbb{N} \land seats \leq 1000$
INITIALISATION
  $seats := 1000$
EVENTS
  $book =$
    WHEN $seats > 0$ **THEN** $seats := seats - 1$ **END**
  $cancel =$
    BEGIN $seats := seats + 1$ **END**
END

A bit less trivial (and consistent) Booking system

CONTEXT
  Booking_context
CONSTANTS
  max_seats
AXIOMS
  $max\_seats \in \mathbb{N}$
  $max\_seats > 0$
END

A bit less trivial (and consistent) Booking system

MACHINE
Booking
SEES
Booking_context
VARIABLES
seats
INVARIANT
$seats \in \mathbb{N} \wedge seats \leq max\_seats$
INITIALISATION
$seats := max\_seats$
EVENTS
$book =$
WHEN $seats > 0$ **THEN** $seats := seats - 1$ **END**
$cancel =$
WHEN $seats < max\_seats$ **THEN** $seats := seats + 1$ **END**
END

Yet another improved Booking system

CONTEXT

Booking_context

CONSTANTS

max_seats, $max\_tickets$

AXIOMS

$max\_seats \in \mathbb{N}$

$max\_seats > 0$

$max\_tickets \in \mathbb{N}1$

$max\_tickets = 5$

END

Yet another improved Booking system

The header stays the same.

MACHINE
Booking
VARIABLES
seats
INVARIANT
$seats \in \mathbb{N} \wedge seats \leq max\_seats$
INITIALISATION
$seats := max\_seats$
...

Yet another improved Booking system (cont.)

The events become parameterised.

$$
\begin{array}{l}
\dots \\
\textbf{EVENTS} \\
\quad \mathit{book} \;=\; \textbf{ANY } n \\
\qquad \textbf{WHERE } n > 0 \;\wedge\; n \leq \mathit{max\_tickets} \;\wedge\; n \leq \mathit{seats} \\
\qquad \textbf{THEN } \mathit{seats} := \mathit{seats} - n \textbf{ END} \\
\quad \mathit{cancel} \;=\; \textbf{ANY } n \\
\qquad \textbf{WHERE } n > 0 \;\wedge\; \mathit{seats} + n \leq \mathit{max\_seats} \\
\qquad \textbf{THEN } \mathit{seats} := \mathit{seats} + n \textbf{ END} \\
\textbf{END}
\end{array}
$$

Event-B mathematical basis: predicate calculus

- A predicate is a logical expression, which can be evaluated to the constants *TRUE* or *FALSE* (of the predefined type *BOOL*), for example, $x \geq y$, $n \geq 0$, $x \in S$, $y \subseteq S$, or $z = Exp(x, y)$
- Standard logical constants and operations in Event-B (graphical notation, followed by the equivalent ascii notation):

$\wedge$	&	logical conjunction
$\vee$	or	logical disjunction
$\Rightarrow$	=>	logical implication
$\Leftrightarrow$	<=>	logical equivalence
$\neg$	not	logical negation
$\forall z.\ P \ \Rightarrow\ Q$	!z. P => Q	universal quantification
$\exists z.\ P \ \wedge\ Q$	#z. P & Q	existential quantification

Mathematical basis: predicate logic

- Predicates are mostly used in operation preconditions (guards), machine invariants, as well as in the constraints on defined sets and constants
- In Event-B tool support (Rodin), the ascii notation will be automatically substituted by the graphical one, once typed
- Rodin will also automatically check the correctness and overall consistency of such formulated properties/constraints by generating so called proof obligations and trying to automatically prove them

Event-B mathematical basis: set theory

- A set is a collection of (non-repeating) entities of some sort
- A set is completely defined by its elements
- Sets can be defined
    - by listing (enumerating) their elements,
    - by specifying properties that characterise their members
- A set can be also abstract, when only its name is given, without revealing anything about the structure of its elements

Some set constants and operations

Graphical notation, followed by the equivalent ascii notation:

$\varnothing$	$\{\}$	empty set
$\{e_1, e_2, ..., e_n\}$	$\{e_1, e_2, ..., e_n\}$	enumerated set
$n_1..n_2$	$n_1..n_2$	interval set between numbers n_1 and n_2
$e \in S$	$e : S$	set membership
$e \notin S$	e /: S	"e does not belong to S", i.e.
$S \subseteq T$	S <: T	set inclusion
$S \nsubseteq T$	S /<: T	"S is not included in T"
$S \cup T$	S \/ T	set union
$S \cap T$	S /\ T	set intersection
$S \setminus T$	S \ T	set difference (subtraction)

Predefined sets like $\mathbb{N}$(*NAT*) for natural numbers, $\mathbb{N}$1(*NAT1*) for positive natural numbers, *BOOL*=$\{$*TRUE*, *FALSE*$\}$ for truth values, etc.

Some other set expressions

$\{z \mid z \in R \land P\}$	Set comprehension: “the subset of R such that P“
$S \times T$ (ascii S**T)	cartesian product $S \times T = \{(x, y) \mid x \in S \land x \in T\}$
$card(S)$	cardinality: the number of set elements
$\mathbb{P}(S)$ (ascii POW(S))	power set: the set of all subsets of S
$\mathbb{P}1(S)$ (ascii POW1(S))	all non-empty subsets of S

A simple example: Request server

Some requirements of the system:

1. The server system handles arrived requests for certain services
2. An arrived requests should be stored in the buffer of received requests
3. Any received request should be inspected before handling
4. After inspection, some of the received (invalid) requests may be rejected without handling
5. If a request is deemed valid, it is handled and put into the buffer of handled requests
6. Any successfully handled request should be acknowledged to the sender
7. ...

A simple example: Request server (cont.)

CONTEXT
Server_context
SETS
REQUESTS

MACHINE Server
SEES Server_context
VARIABLES
received, *handled*, *completed*
INVARIANT
$received \subseteq \text{REQUEST} \wedge handled \subseteq \text{REQUEST} \wedge$
$completed \subseteq \text{REQUEST} \wedge completed \subseteq handled \wedge$
$handled \subseteq received$
INITIALISATION
$received,\ handled,\ completed := \varnothing, \varnothing, \varnothing$

...

Abstract sets

- Note that the definition of the set REQUESTS is abstract
- Only the name of the set is introduced, without giving any other details about the structure of its elements
- Such a set can be considered as a model parameter, so that any concrete set can be used instead
- If we need to assume some things about this set, these assumptions should be formulated as axioms, e.g., $\mathrm{REQUESTS} \neq \varnothing$
(the set should be non-empty)

Example: Request server (cont.)

$$
\begin{array}{l}
\textsf{EVENTS} \\
\quad \mathit{receive} \;=\; \textsf{ANY}\ rr \\
\qquad \textsf{WHERE}\ rr \in \mathit{REQUEST} \\
\qquad \textsf{THEN}\ \mathit{received} := \mathit{received} \cup \{rr\}\ \textsf{END} \\
\quad \mathit{handle} \;=\; \textsf{ANY}\ rr \\
\qquad \textsf{WHERE}\ rr \in \mathit{received} \\
\qquad \textsf{THEN}\ \mathit{handled} := \mathit{handled} \cup \{rr\}\ \textsf{END} \\
\quad \mathit{reject} \;=\; \textsf{ANY}\ rr \\
\qquad \textsf{WHERE}\ rr \in \mathit{received}\ \wedge\ rr \notin \textsf{handled} \\
\qquad \textsf{THEN}\ \mathit{received} := \mathit{received} \setminus \{rr\}\ \textsf{END} \\
\quad \mathit{acknowledge} \;=\; \textsf{ANY}\ rr \\
\qquad \textsf{WHERE}\ rr \in \mathit{handled} \\
\qquad \textsf{THEN}\ \mathit{completed} := \mathit{completed} \cup \{rr\}\ \textsf{END} \\
\textsf{END}
\end{array}
$$

An example for the exercise session: a vending machine

1. The vending machine serves the customer a product from a number of available choices;
2. The customer may select one of the available product choices;
3. After selection, the customer may proceed by paying for the selected product or cancelling the selection;
4. After sufficient payment, the selected product is served to the customer;
5. The payment is conducted by entering money into the machine;
6. The available product choices and their prices may be updated by the machine operator.

Example: a vending machine (cont.)

7. The selected product can be only served after the payment is made;
8. Once the customer is served or he/she cancels the service, the selection is dropped, i.e., becomes NONE;
9. Once the selection for the previous customer is dropped, the machine gets ready to serve a new customer;
10. The vending machine is ready to serve a new customer if and only if the previous customer has been served.

Example: a vending machine (cont.)

An expected system usecase: