Applications of Formal Verification

Formal Software Design: Modelling in Event-B

Bernhard Beckert · Mattias Ulbrich | SS 2017
Jean-Raymond Abrial: 
*Modelling in Event-B*
System and Software Engineering
Cambridge University Press, 2010

Jean-Raymond Abrial: 
*The B-Book:*
Assigning programs to meanings
Cambridge University Press, 1996
Abstraction and Refinement – Introduction
Late fault recovery is expensive

Late fault recovery is expensive


Goal: Detect faults here!
Reasons for system faults

- Systems are inherently complex
- Unconsidered situations, corner cases
- Ambiguous natural language requirements
- Component interplay
- ...
Reasons for system faults

- Systems are inherently complex
- Unconsidered situations, corner cases
- Ambiguous natural language requirements
- Component interplay
- ...
Reasons for system faults

- Systems are inherently complex
- Unconsidered situations, corner cases
- Ambiguous natural language requirements
- Component interplay
- ...
Reasons for system faults

- Systems are inherently complex
- Unconsidered situations, corner cases
- Ambiguous natural language requirements
- Component interplay
- ...
Reasons for system faults

- Systems are inherently complex
- Unconsidered situations, corner cases
- Ambiguous natural language requirements
- Component interplay
- ...
Abstraction

The only tool to master complexity is abstraction.

Cliff Jones
Abstraction and Refinement
Abstraction and Refinement

Abstraction

Concrete

Refinement

Abstraction
Abstraction and Refinement
Abstraction and Refinement

Abstract

Concrete

Refinement

Abstraction
### Abstraction
- reduce system complexity
- without removing important properties
- make the model susceptible to formal analysis

and the inverse

### Refinement
- enrich abstract model with details
- introduce a new particular aspect
- iterative process: add complexity in a stepwise fashion
Abstraction is an important tool in engineering

Established means of abstraction

- Mechanical engineering: BLUEPRINTS
- Electrical engineering: DATASHEETS
- CIRCUIT DIAGRAMS
- Architecture: FLOOR PLANS
- ...

Abstract descriptions remove unnecessary details, concentrate on one aspect
Datasheet – Abstraction

Extracts from datasheet for an IC with four NAND gates
Datasheet – Abstraction

Extracts from datasheet for an IC with four NAND gates

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect</td>
<td>Geometry</td>
</tr>
</tbody>
</table>

Beckert, Ulbrich – Applications of Formal Verification

SS 2017 10/96
Datasheet – Abstraction

Extracts from datasheet for an IC with four NAND gates

Aspect Behaviour

Aspect Geometry

refined to
Datasheet – Abstraction

Extracts from datasheet for an IC with four NAND gates

Aspect Behaviour

refined to

Aspect Geometry

refined to

Beckert, Ulbrich – Applications of Formal Verification
Schematic Diagram vs. PCB Layout

Arduino™ UNO Rev3
Schematic Diagram vs. PCB Layout

Aspect
“Behaviour” preserved
Beck diagrams (1931)
Beck diagrams (1931)

Aspect “Route planning” is preserved
Property preservation

Abstraction with focus on particular aspect

System properties w.r.t. that aspect must also hold in the abstraction.

Refinement with focus on particular aspect

Properties of abstract model w.r.t. that aspect must be inherited by the refined model.

Examples:

- **Abstraction**: “The shortest tube travel from Liverpool St. to Westminster has 8 stops and 2 changes.”
- **Refinement**: *Abstract*: Input “\(a = 1\)” gives output “\(b = 1\)”
  *Concrete*: High voltage on pin A gives high voltage on pin B
Property preservation

Abstraction with focus on particular aspect
System properties w.r.t. that aspect must also hold in the abstraction.

Refinement with focus on particular aspect
Properties of abstract model w.r.t. that aspect must be inherited by the refined model.

That’s what we will formally prove in the next sections.

Examples:

- **Abstraction**: “The shortest tube travel from Liverpool St. to Westminster has 8 stops and 2 changes.”
- **Refinement**: 
  - Abstract: Input “a = 1” gives output “b = 1”
  - Concrete: High voltage on pin A gives high voltage on pin B
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

Abstraction as a technique

That’s what we will look into in the next sections.
# “Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

## Conceptual abstraction
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

## Abstraction as a technique
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

Abstraction as a technique
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

**Conceptual abstraction**
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

**Abstraction as a technique**

That's what we will look into in the next sections.
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

**Conceptual abstraction**
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

**Abstraction as a technique**
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

**Conceptual abstraction**
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

**Abstraction as a technique**
### “Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

<table>
<thead>
<tr>
<th>Conceptual abstraction</th>
<th>Abstraction as a technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>- reduce complexity for more comprehensibility</td>
<td>- reduce complexity to enhance performance/reach of a tool</td>
</tr>
<tr>
<td>- focus on a particular system aspect</td>
<td>- abstract from given predicates to uninterpreted predicates</td>
</tr>
<tr>
<td>- provided by designer/developer</td>
<td>- computed automatically</td>
</tr>
<tr>
<td>- refinement introduces new aspect</td>
<td>- refinement driven by failed proofs</td>
</tr>
</tbody>
</table>

(Counter-Example Guided Abstraction Refinement, CEGAR)
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

**Conceptual abstraction**
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

**Abstraction as a technique**
- reduce complexity to enhance performance/reach of a tool
- abstract from given predicates to uninterpreted predicates
- computed automatically
- refinement driven by failed proofs
  (Counter-Example Guided Abstraction Refinement, CEGAR)
Two areas of abstraction and refinement in formal methods:

**Conceptual abstraction**
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

Abstraction as a technique
- reduce complexity to enhance performance/reach of a tool
- abstract from given predicates to uninterpreted predicates
- computed automatically
- refinement driven by failed proofs (Counter-Example Guided Abstraction Refinement, CEGAR)
“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

**Conceptual abstraction**
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

**Abstraction as a technique**
- reduce complexity to enhance performance/reach of a tool
- abstract from given predicates to uninterpreted predicates
- **computed automatically**
- refinement driven by failed proofs
  (Counter-Example Guided Abstraction Refinement, CEGAR)
"Conceptual" vs "Technical" Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction
- reduce complexity for more comprehensibility
- focus on a particular system aspect
- provided by designer/developer
- refinement introduces new aspect

Abstraction as a technique
- reduce complexity to enhance performance/reach of a tool
- abstract from given predicates to uninterpreted predicates
- computed automatically
- refinement driven by failed proofs
  (Counter-Example Guided Abstraction Refinement, CEGAR)
### “Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

<table>
<thead>
<tr>
<th>Conceptual abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>- reduce complexity for more comprehensibility</td>
</tr>
<tr>
<td>- focus on a particular system aspect</td>
</tr>
<tr>
<td>- provided by designer/developer</td>
</tr>
<tr>
<td>- refinement introduces new aspect</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Abstraction as a technique</th>
</tr>
</thead>
<tbody>
<tr>
<td>- reduce complexity to enhance performance/reach of a tool</td>
</tr>
<tr>
<td>- abstract from given predicates to uninterpreted predicates</td>
</tr>
<tr>
<td>- computed automatically</td>
</tr>
<tr>
<td>- refinement driven by failed proofs (Counter-Example Guided Abstraction Refinement, CEGAR)</td>
</tr>
</tbody>
</table>

That’s what we will look into in the next sections.
Event-B

- EventB is a formalism for modelling and reasoning about discrete systems.
  - for their structure (how can their state be described) and
  - for their behaviour (how can the evolution of their state be described)

- Models are formulated using set theory

- Event-based evolution of the original B Method

- Tool-support:
  - RODIN – deductive verification, theorem prover: proofs
  - Pro-B – model checking, animator: counterexamples
Central Concepts

- **Variables and Events**
  - *Variables* model the current state within the state space.
  - *Events* describe operations to model the system behaviour.

- **Invariants**
  - properties to be maintained by system
  - formal proof obligations to show that

- **Refinement**
  - Behaviour of refining model is compatible with abstract model
  - formal proof obligation to show that
  - Hence, invariants of abstract model are inherited by concrete model
Central Concepts

- **Variables and Events**
  - *Variables* model the current state within the state space.
  - *Events* describe operations to model the system behaviour

- **Invariants**
  - properties to be maintained by system
  - formal proof obligations to show that

- **Refinement**
  - Behaviour of refining model is compatible with abstract model
  - formal proof obligation to show that
  - Hence, invariants of abstract model are inherited by concrete model
Event-B models consist of **contexts and machines**:

**Contexts**
- **Static, rigid, constant** parts that *do not* change over time.

**Machines**
- **Dynamic, volatile, evolving** parts that *do* change over time.
## Contexts and Machines

Event-B models consist of **contexts and machines**:

### Contexts

- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines

- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
## Contexts and Machines

Event-B models consist of **contexts and machines**:

### Contexts
- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines
- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
## Contexts and Machines

Event-B models consist of **contexts and machines**:

### Contexts

- *Carrier sets* (ground types, universes, “urelements”)
- *Constants* (state-independent symbols, rigid symbols)
- *Axioms* (formulas valid by stipulation)
- *Theorems* (formulas proved valid)

### Machines

- *Context references* (which symbols are available)
- *Variables* (state-dependent symbols, non-rigid symbols, program variables)
- *Invariants* (formulas true in every reachable system state)
- *Events* (state transition descriptions)

(Explanations or alternative names in parens)
## Contexts and Machines

Event-B models consist of **contexts and machines**:

### Contexts
- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines
- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
## Contexts and Machines

Event-B models consist of **contexts and machines**: 

### Contexts
- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines
- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
**Contexts and Machines**

Event-B models consist of **contexts and machines**:

### Contexts
- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines
- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
## Contexts and Machines

Event-B models consist of **contexts and machines**:

### Contexts
- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines
- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
Event-B models consist of **contexts and machines**:

### Contexts
- **Carrier sets** (ground types, universes, “urelements”)
- **Constants** (state-independent symbols, rigid symbols)
- **Axioms** (formulas valid by stipulation)
- **Theorems** (formulas proved valid)

### Machines
- **Context references** (which symbols are available)
- **Variables** (state-dependent symbols, non-rigid symbols, program variables)
- **Invariants** (formulas true in every reachable system state)
- **Events** (state transition descriptions)

(Explanations or alternative names in parens)
Students and Exams – Requirements

R1 Every student must take a final exam in a subject of their choice.

R2 They can have attempts without yet failing or passing.

R3 Eventually they can pass or fail, but never both.

→ Identify the context, the state and the events according to the requirements R1–R3.
Introduction by Example

Students and Exams – Requirements

R1 Every **student** must take a final exam in a **subject** of their choice.

R2 They can have **attempts** without yet failing or passing.

R3 Eventually they can **pass** or **fail**, but **never both**.

→ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.
# Introduction by Example

## Students and Exams – Requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Every <strong>student</strong> must take a final exam in a <strong>subject</strong> of their choice.</td>
</tr>
<tr>
<td>R2</td>
<td>They can have <strong>attempts</strong> without yet failing or passing.</td>
</tr>
<tr>
<td>R3</td>
<td>Eventually they can <strong>pass</strong> or <strong>fail</strong>, but <strong>never both</strong>.</td>
</tr>
</tbody>
</table>

→ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.
Students and Exams – Requirements

R1 Every student must take a final exam in a subject of their choice.

R2 They can have attempts without yet failing or passing.

R3 Eventually they can pass or fail, but never both.

→ Identify the context, the state and the events according to the requirements R1–R3.
Students and Exams – Requirements

R1 Every *student* must take a final exam in a *subject* of their choice.

R2 They can have *attempts* without yet failing or passing.

R3 Eventually they can *pass* or *fail*, but never both.

→ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.
Introduction by Example

Students and Exams – Requirements

R1 Every student must take a final exam in a subject of their choice.

R2 They can have attempts without yet failing or passing.

R3 Eventually they can pass or fail, but never both.

Identify the context, the state and the events according to the requirements R1–R3.
CONTEXTEXamCtxt
CONTEXT ExamCtxt

SETS

STUDENT // see requirement R1
SUBJECT
Exam Context

CONTEXT \( \text{ExamCtx} \)

SETS

\( \text{STUDENT} \) // see requirement R1

\( \text{SUBJECT} \)

CONSTANTS

\( \text{maths} \quad \text{physics} \quad \text{siblings} \)
CONTEXT ExamCtxt

SETS
STUDENT // see requirement R1
SUBJECT

CONSTANTS
maths physics siblings

AXIOMS
maths ∈ SUBJECT // type of variables
physics ∈ SUBJECT
Exam Context

CONTEXT ExamCtxt

SETS
STUDENT // see requirement R1
SUBJECT

CONSTANTS
maths physics siblings

AXIOMS
maths ∈ SUBJECT // type of variables
physics ∈ SUBJECT
maths ≠ physics // constants could have same value
CONTEXT \textit{ExamCtxt}

SETS
\begin{itemize}
\item \textsc{Student} \quad // \text{see requirement R1}
\item \textsc{Subject}
\end{itemize}

CONSTANTS
\begin{itemize}
\item maths
\item physics
\item siblings
\end{itemize}

AXIOMS
\begin{itemize}
\item maths \in \textsc{ Subject} \quad // \text{type of variables}
\item physics \in \textsc{ Subject}
\item maths \neq physics \quad // \text{constants could have same value}
\item siblings \subseteq \textsc{Student} \times \textsc{Student} \quad // \text{function type}
\end{itemize}
Exam Context

CONTEXT ExamCtxt

SETS

STUDENT // see requirement R1
SUBJECT

CONSTANTS

maths physics siblings

AXIOMS

maths \in SUBJECT // type of variables
physics \in SUBJECT
maths \neq physics // constants could have same value
siblings \subseteq STUDENT \times STUDENT // function type
\forall s \cdot s \in STUDENT \Rightarrow (s \mapsto s) \notin siblings // irreflexive
// ...

Beckert, Ulbrich – Applications of Formal Verification
MACHINE ExamAbstract
MACHINE ExamAbstract
SEES ExamCtxt
Exam Machine

MACHINE ExamAbstract
SEES ExamCtxt

VARIABLES
    passed   failed
Exam Machine

MACHINE ExamAbstract
SEES ExamCtxt

VARIABLES
  passed    failed

INVARIANTS
  passed ⊆ STUDENT    failed ⊆ STUDENT
MACHINE ExamAbstract
SEES ExamCtxt

VARIABLES
  passed    failed

INVARIANTS
  passed ⊆ STUDENT   failed ⊆ STUDENT
  passed ∩ failed = ∅  // R3
Exam Machine

MACHINE ExamAbstract
SEES ExamCtxt

VARIABLES
  passed  failed

INVARIANTS
  passed ⊆ STUDENT  failed ⊆ STUDENT
  passed ∩ failed = ∅  // R3

EVENTS
  INITIALISATION ≜ . . .
  ATTEMPT EXAM ≜ . . .  // R2
  PASS EXAM ≜ . . .  // R3
  FAIL EXAM ≜ . . .  // R3
Exam Machine (2)

MACHINE ExamAbstract
VARIABLES passed failed . . .

EVENTS

INITIALISATION \( \hat{=} \)

\[
\text{failed} \; := \; \emptyset \\
\text{passed} \; := \; \emptyset
\]
Exam Machine (2)

MACHINE ExamAbstract
VARIABLES passed failed . . .

EVENTS
INITIALISATION ≜
  failed := ∅
  passed := ∅

PASSEXAM ≜
  ANY s grade
  WHERE s ∈ STUDENT ∧ grade ≤ 4
  THEN passed := passed ∪ {s}
MACHINE ExamAbstract
VARIABLES passed failed 

EVENTS
INITIALISATION ⇒
   failed := ∅
   passed := ∅

PASSEXAM ⇒
   ANY s grade
   WHERE s ∈ STUDENT ∧ grade ≤ 4
   THEN passed := passed ∪ {s}

FAILEXAM ⇒
   ANY s grade
   WHERE s ∈ STUDENT ∧ grade > 4
   THEN failed := failed ∪ {s}
MACHINE ExamAbstract
VARIABLES passed failed
INVARINTS passed ∩ failed = ∅ ... 

EVENTS
PASSEXAM ≜
  ANY s grade
  WHERE s ∈ STUDENT ∧ grade ≤ 4
  THEN passed := passed ∪ {s}

FAILEXAM ≜
  ANY s grade
  WHERE s ∈ STUDENT ∧ grade > 4
  THEN failed := failed ∪ {s}
Invariant violated

MACHINE ExamAbstract
VARIABLES passed failed
INVARINTS passed $\cap$ failed = $\emptyset$ ...

EVENTS
PASS EXAM $\triangleq$
  ANY $s$ grade
  WHERE $s \in$ STUDENT $\setminus$ (failed $\cup$ passed) $\land$ grade $\leq$ 4
  THEN passed $\leftarrow$ passed $\cup \{s\}$

FAIL EXAM $\triangleq$
  ANY $s$ grade
  WHERE $s \in$ STUDENT $\setminus$ (failed $\cup$ passed) $\land$ grade $>$ 4
  THEN failed $\leftarrow$ failed $\cup \{s\}$
Unspecifiead model

EVENTS

\textsc{passExam} \triangleq
\begin{align*}
\text{ANY } s & \text{ grade WHERE grade } \leq 4 \land s \in \ldots \\
\text{THEN } passed & \colon= \text{passed } \cup \{ s \}
\end{align*}

\textsc{failExam} \triangleq
\begin{align*}
\text{ANY } s & \text{ grade WHERE grade } > 4 \land s \in \ldots \\
\text{THEN } failed & \colon= \text{failed } \cup \{ s \}
\end{align*}

\textsc{attemptExam} \triangleq
\begin{align*}
\text{ANY } s & \text{ grade WHERE grade } \in \mathbb{N} \land s \in \ldots \\
\text{THEN } skip & \colon= \text{skip}
\end{align*}
**Underspecified model**

**EVENTS**

\[ \text{PASS\textsc{EXAM} } \triangleq \]

\[
\text{ANY } s \text{ grade WHERE } \text{grade} \leq 4 \land s \in \ldots \\
\text{THEN passed} := \text{passed} \cup \{s\}
\]

\[ \text{FAIL\textsc{EXAM} } \triangleq \]

\[
\text{ANY } s \text{ grade WHERE } \text{grade} > 4 \land s \in \ldots \\
\text{THEN failed} := \text{failed} \cup \{s\}
\]

\[ \text{ATTEMPT\textsc{EXAM} } \triangleq \]

\[
\text{ANY } s \text{ grade WHERE } \text{grade} \in \mathbb{N} \land s \in \ldots \\
\text{THEN skip}
\]

**Additional requirement**

**R4** Any student may attempt the exam three times and ultimately fails if the fourth attempt is unsuccessful.
Refinement Exams (1)

MACHINE RefinedExams REFINES ExamsAbstract
Refinement Exams (1)

MACHINE RefinedExams REFINES ExamsAbstract
VARIABLES passed attempts
MACHINE RefinedExams REFINES ExamsAbstract
VARIABLES passed attempts
INVARIANTS

\[
\text{attempts} \in \text{STUDENT} \rightarrow \mathbb{N} \quad \text{// typing for attempts}
\]
\[
\text{failed} = \{ s \cdot \text{attempts}(s) = 4 \} \quad \text{// coupling invariant}
\]
MACHINE RefineExams REFINES ExamsAbstract
VARIABLES passed attempts
INVARIANTS
\[ \text{attempts} \in \text{STUDENT} \rightarrow \mathbb{N} \] // typing for attempts
\[ \text{failed} = \{ s \cdot \text{attempts}(s) = 4 \} \] // coupling invariant
EVENTS
INITIALISATION \[ \triangleq \text{REFINES INITIALISATION} \]
\[ \text{passed} := \emptyset \]
\[ \text{attempts} := \{ s \cdot s \in \text{STUDENT} \mid (s \mapsto 0) \} \]
MACHINE *RefinedExams* REFINES *ExamsAbstract*

VARIABLES *passed attempts*

INVARIANTS

\[
\text{attempts} \in STUDENT \rightarrow \mathbb{N} \quad // \text{typing for attempts}
\]

\[
\text{failed} = \{ s \cdot \text{attempts}(s) = 4 \} \quad // \text{coupling invariant}
\]

EVENTS

INITIALISATION \( \triangleq \) REFINES INITIALISATION

\[
\text{passed} := \emptyset
\]

\[
\text{attempts} := \{ s \cdot s \in STUDENT \mid (s \mapsto 0) \}
\]

EXAMULTIMATEFAIL \( \triangleq \) REFINES EXAMFAIL . . .

EXAMMISSED \( \triangleq \) REFINES EXAMATTEMPT . . .

EXAMPASSED \( \triangleq \) REFINES EXAMPASSED . . .
EVENTS

\[ \text{EXAM}_{\text{ULTIMATE}} \text{FAIL} \equiv \text{REFINES} \text{ EXAM}_{\text{FAIL}} \]
\[ \text{ANY } s \text{ grade} \]
\[ \text{WHERE } \ldots \land grade > 4 \land \text{attempts}(s) = 3 \]
\[ \text{THEN} \]
\[ \text{attempts}(s) := \text{attempts}(s) + 1 \]

\[ \text{EXAM}_{\text{MISSED}} \equiv \text{REFINES} \text{ EXAM}_{\text{ATTEMPT}} \]
\[ \text{ANY } s \text{ grade} \]
\[ \text{WHERE } \ldots \land grade > 4 \land \text{attempts}(s) < 3 \]
\[ \text{THEN} \]
\[ \text{attempts}(s) := \text{attempts}(s) + 1 \]

\[ \ldots \]
Refinement Exams (3)

This refinement takes now also R4 into account.

Refinement preserves invariants

1. Every possible event of *RefinedExams* is a possible event in *ExamsAbstract*

2. Every invariant of *ExamsAbstract* is also an invariant of *RefinedExams*

We will come back to this more formally ...
Set Theory –
Equipment for formal modelling
Set theory – a universal modelling language

Not only used in Event-B.

Set theory also used for modelling in ...

- Z
- Object-Z, Z++
- (classical) B
- Event-B
- Alloy
- ...
Every term in Event-B has a unique type.

Types are *part of the syntax* of Event-B and some expressions are syntactically forbidden:

\[ \text{maths} \in \text{failed} \quad \text{is syntactically invalid.} \]

(remember: \( \text{math} \in \text{SUBJECT} \), \( \text{failed} \subseteq \text{STUDENT} \))

“You can’t compare apples and oranges.”
Set Theory

Formal language in Event-B models

Typed First Order Set Theory with Additional Theories

- sets are objects in the logic
- first order axioms define the semantics of sets
- quantification over sets is allowed
- quantification over predicates, functions is not allowed
- (Foundation is a typed Zermelo-Fraenkel axiomatisation)
Set Theory

Formal language in Event-B models

Typed **First Order Set Theory** with Additional Theories

- sets are objects in the logic
- first order axioms define the semantics of sets
- quantification over sets is allowed
- quantification over predicates, functions is not allowed
- (Foundation is a typed Zermelo-Fraenkel axiomatisation)
Set Theory

Formal language in Event-B models

Typed **First Order Set Theory** with Additional Theories

- sets are objects in the logic
- first order axioms define the semantics of sets
- quantification over sets is allowed
- quantification over predicates, functions is not allowed
- (Foundation is a typed Zermelo-Fraenkel axiomatisation)
Set Theory

Formal language in Event-B models

Typed **First Order Set Theory** with Additional Theories

- sets are objects in the logic
- first order axioms define the semantics of sets
- quantification over sets is allowed
- quantification over predicates, functions is not allowed
- (Foundation is a typed Zermelo-Fraenkel axiomatisation)
Set Theory

Formal language in Event-B models

Typed **First Order Set Theory** with Additional Theories

- sets are objects in the logic
- first order axioms define the semantics of sets
- quantification over sets is allowed
- quantification over predicates, functions is not allowed
- (Foundation is a typed Zermelo-Fraenkel axiomatisation)
There are additional theories with fixed semantics
- integers
- more theories (datatypes) can be added by user (an extension to the system)
Types

1. BOOL and \( \mathbb{Z} \) are types.

2. Every carrier set declared in a CONTEXT is a type.

3. If \( T \) is a type then \( \mathcal{P}(T) \) is a type.
   Semantics: \( \mathcal{P}(T) \) is the set of all subsets of \( T \) (powerset).

4. If \( T_1, T_2 \) are types then \( T_1 \times T_2 \) is a type.
   Semantics: \( T_1 \times T_2 \) is the set of all ordered pairs \((a, b)\) with \( a \in T_1 \) and \( b \in T_2 \) (Cartesian product).

Every expression \( E \) has a unique type \( \tau(E) \).
Types

1. **BOOL** and **Z** are types

2. Every carrier set declared in a **CONTEXT** is a type.

3. If **T** is a type then \( \mathbb{P}(T) \) is a type.
   *Semantics:* \( \mathbb{P}(T) \) is the set of all subsets of **T** (powerset).

4. If **T**₁, **T**₂ are types then \( T_1 \times T_2 \) is a type.
   *Semantics:* \( T_1 \times T_2 \) is the set of all ordered pairs \((a, b)\) with \( a \in T_1 \) and \( b \in T_2 \) (Cartesian product).

Every expression **E** has a unique type \( \tau(E) \).
Types

1. BOOL and \( \mathbb{Z} \) are types

2. Every carrier set declared in a CONTEXT is a type.

3. If \( T \) is a type then \( \mathcal{P}(T) \) is a type.
   \textbf{Semantics:} \( \mathcal{P}(T) \) is the set of all subsets of \( T \) (powerset).

4. If \( T_1, T_2 \) are types then \( T_1 \times T_2 \) is a type.
   \textbf{Semantics:} \( T_1 \times T_2 \) is the set of all ordered pairs \((a, b)\) with \( a \in T_1 \) and \( b \in T_2 \) (Cartesian product).

Every expression \( E \) has a unique type \( \tau(E) \).
Types

1. BOOL and \( \mathbb{Z} \) are types

2. Every carrier set declared in a CONTEXT is a type.

3. If \( T \) is a type then \( \mathcal{P}(T) \) is a type.
   **Semantics:** \( \mathcal{P}(T) \) is the set of all subsets of \( T \) (powerset).

4. If \( T_1, T_2 \) are types then \( T_1 \times T_2 \) is a type.
   **Semantics:** \( T_1 \times T_2 \) is the set of all ordered pairs \((a, b)\) with \( a \in T_1 \) and \( b \in T_2 \) (Cartesian product).

Every expression \( E \) has a unique type \( \tau(E) \).
Types

1. BOOL and ℤ are types

2. Every carrier set declared in a CONTEXT is a type.

3. If \( T \) is a type then \( \mathbb{P}(T) \) is a type.
   **Semantics:** \( \mathbb{P}(T) \) is the set of all subsets of \( T \) (powerset).

4. If \( T_1, T_2 \) are types then \( T_1 \times T_2 \) is a type.
   **Semantics:** \( T_1 \times T_2 \) is the set of all ordered pairs \((a, b)\) with \( a \in T_1 \) and \( b \in T_2 \) (Cartesian product).

Every expression \( E \) has a unique type \( \tau(E) \).
Set theory needs not be typed: Everything can be viewed as a set.

Reasons to introduce types:

- some specification errors may be detected as syntax errors (even before the verification has started)
- avoid Russell’s paradox
Types (2)

Set theory needs not be typed: Everything can be viewed a set.

Reasons to introduce types:
- some specification errors may be detected as syntax errors (even before the verification has started)
- avoid Russell’s paradox

Russell’s paradox
Assume that the expression \(\{ s \mid \phi \}\) for any formula \(\phi\) denotes a set. Let \(R := \{ s \mid s \notin s\}\).
Set theory needs not be typed: Everything can be viewed a set.

**Reasons to introduce types:**
- some specification errors may be detected as syntax errors (even before the verification has started)
- avoid Russell’s paradox

**Russell’s paradox**
Assume that the expression \( \{ s \mid \phi \} \) for any formula \( \phi \) denotes a set. Let \( R := \{ s \mid s \notin s \} \).
One observes: \( R \in R \iff R \notin R \)
\( \perp \)
Set theory needs not be typed: Everything can be viewed a set.

**Reasons to introduce types:**

- some specification errors may be detected as syntax errors (even before the verification has started)
- avoid Russell’s paradox

**Russell’s paradox**

Assume that the expression $\{ s \mid \phi \}$ for any formula $\phi$ denotes a set. Let $R := \{ s \mid s \notin s \}$. Not allowed with types.

One observes: $R \in R \iff R \notin R \downarrow$

*(This crushed naive set theory in early 1900s.)*
Sets

Constructors for sets:

- empty set $\emptyset : \mathcal{P}(S)$
Constructors for sets:

- empty set $\emptyset : \mathcal{P}(S)$
- set extension $\{ \ldots \} : S^* \rightarrow \mathcal{P}(S)$
  
  example: $\{1, 2\} : \mathcal{P}(\mathbb{Z})$
Sets

Constructors for sets:

- empty set $\emptyset : \mathcal{P}(S)$
- set extension $\{ \ldots \} : S^* \rightarrow \mathcal{P}(S)$
  example: $\{1, 2\} : \mathcal{P}(\mathbb{Z})$
- carrier sets $C : \mathcal{P}(C)$
  example: $\text{STUDENT} : \mathcal{P}(\text{STUDENT})$
Constructors for sets:

- empty set $\emptyset : \mathcal{P}(S)$
- set extension $\{ \ldots \} : S^* \rightarrow \mathcal{P}(S)$
  example: $\{1, 2\} : \mathcal{P}(\mathbb{Z})$
- carrier sets $C : \mathcal{P}(C)$
  example: $STUDENT : \mathcal{P}(STUDENT)$
- powerset $\mathcal{P}(\cdot) : \mathcal{P}(S) \rightarrow \mathcal{P}(\mathcal{P}(S))$
  example: $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} : \mathcal{P}(\mathbb{Z})$
Constructors for sets:

- empty set $\emptyset : \mathcal{P}(S)$
- set extension $\{ \ldots \} : S^* \rightarrow \mathcal{P}(S)$
  - example: $\{1, 2\} : \mathcal{P}(\mathbb{Z})$
- carrier sets $C : \mathcal{P}(C)$
  - example: $\text{STUDENT} : \mathcal{P}(\text{STUDENT})$
- powerset $\mathcal{P}(\cdot) : \mathcal{P}(S) \rightarrow \mathcal{P}(\mathcal{P}(S))$
  - example: $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} : \mathcal{P}(\mathbb{Z})$
- product $\cdot \times \cdot : \mathcal{P}(S) \times \mathcal{P}(T) \rightarrow \mathcal{P}(S \times T)$
  - example: $\text{BOOL} \times \{1\} = \{\{\text{true}, 1\}, \{\text{false}, 1\}\} : \mathcal{P}(\text{BOOL} \times \mathbb{Z})$
Sets

Constructors for sets:

- empty set $\emptyset : \mathcal{P}(S)$
- set extension $\{ \ldots \} : S^* \rightarrow \mathcal{P}(S)$
  example: $\{1, 2\} : \mathcal{P}(\mathbb{Z})$
- carrier sets $C : \mathcal{P}(C)$
  example: $STUDENT : \mathcal{P}(STUDENT)$
- powerset $\mathcal{P}(\cdot) : \mathcal{P}(S) \rightarrow \mathcal{P}(\mathcal{P}(S))$
  example: $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} : \mathcal{P}(\mathbb{Z})$
- product $\cdot \times \cdot : \mathcal{P}(S) \times \mathcal{P}(T) \rightarrow \mathcal{P}(S \times T)$
  example: $BOOL \times \{1\} = \{\{true, 1\}, \{false, 1\}\} : \mathcal{P}(BOOL \times \mathbb{Z})$
- set comprehension $\{x \cdot \varphi \mid e\}$
  formula $\varphi$, term $e : T$, result of type $\mathcal{P}(T)$
  example: $\{x \cdot x \geq 2 \mid x \ast x\} = \{4, 9, 16, \ldots\}$
Relations

- Relations are sets of pairs (tuples).
Relations

- Relations are sets of pairs (tuples).
- All relations: \(E_1 \leftrightarrow E_2 := \mathcal{P}(E_1 \times E_2)\)
Relations

- Relations are sets of pairs (tuples).
- All relations: $E_1 \leftrightarrow E_2 := \mathcal{P}(E_1 \times E_2)$
- Pairs $(E_1 \leftrightarrow E_2) : \tau(E_1) \times \tau(E_2)$
Relations

- Relations are sets of pairs (tuples).
- All relations: $E_1 \leftrightarrow E_2 := \mathcal{P}(E_1 \times E_2)$
- Pairs $(E_1 \leftrightarrow E_2) : \tau(E_1) \times \tau(E_2)$
- Domain of a relation $\text{dom}(R)$
  \[ \text{dom}(R) = \{ x, y : (x \leftrightarrow y) \in R \mid x \} \]
  example: $\text{dom}(E_1 \times E_2) = E_1$
Relations

- Relations are sets of pairs (tuples).

- All relations: $E_1 \leftrightarrow E_2 := \mathbb{P}(E_1 \times E_2)$

- Pairs $(E_1 \leftrightarrow E_2) : \tau(E_1) \times \tau(E_2)$

- Domain of a relation $\text{dom}(R)$
  
  $\text{dom}(R) = \{x, y : (x \mapsto y) \in R \mid x\}$

  example: $\text{dom}(E_1 \times E_2) = E_1$

- Range of a relation $\text{ran}(R)$
  
  $\text{ran}(R) = \{x, y : (x \mapsto y) \in R \mid y\}$

  example: $\text{ran}(E_1 \times E_2) = E_2$
Relations

- Relations are sets of pairs (tuples).
- All relations: $E_1 \leftrightarrow E_2 := \mathcal{P}(E_1 \times E_2)$
- Pairs $(E_1 \leftrightarrow E_2) : \tau(E_1) \times \tau(E_2)$
- Domain of a relation $\text{dom}(R)$
  \[ \text{dom}(R) = \{ x, y : (x \leftrightarrow y) \in R | x \} \]
  example: $\text{dom}(E_1 \times E_2) = E_1$
- Range of a relation $\text{ran}(R)$
  \[ \text{ran}(R) = \{ x, y : (x \leftrightarrow y) \in R | y \} \]
  example: $\text{ran}(E_1 \times E_2) = E_2$
- can be nested: $(E_1 \leftrightarrow E_2) \leftrightarrow E_3$ for a ternary relation etc.
Relations

- Relations are sets of pairs (tuples).
- All relations: \( E_1 \leftrightarrow E_2 := \mathcal{P}(E_1 \times E_2) \)
- Pairs \((E_1 \leftrightarrow E_2) : \tau(E_1) \times \tau(E_2)\)
- **Domain** of a relation \(\text{dom}(R)\)
  \[ \text{dom}(R) = \{ x, y \cdot (x \leftrightarrow y) \in R \mid x \} \]
  example: \(\text{dom}(E_1 \times E_2) = E_1 \) if \(E_2 \neq \emptyset\)
- **Range** of a relation \(\text{ran}(R)\)
  \[ \text{ran}(R) = \{ x, y \cdot (x \leftrightarrow y) \in R \mid y \} \]
  example: \(\text{ran}(E_1 \times E_2) = E_2 \) if \(E_1 \neq \emptyset\)
- can be nested: \((E_1 \leftrightarrow E_2) \leftrightarrow E_3\) for a ternary relation etc.