Applications of Formal Verification

Formal Software Design: Modelling in Event-B

Dr. Vladimir Klebanov · Dr. Mattias Ulbrich | SS 2015
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

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 and Refinement

Abstraction

Refinement

Abstraction
Abstraction and Refinement
Abstraction and Refinement

- Abstract
- Concrete

Abstraction and Refinement: the process of moving from an abstract model to a more concrete implementation and vice versa.
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>
</table>

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Geometry</th>
</tr>
</thead>
</table>

Klebanov, Ulbrich – Applications of Formal Verification
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

 Aspect Geometry

Klebanov, 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**

- Reduce complexity for more comprehensibility
- Focus on a particular system aspect provided by designer/developer

**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)

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**
“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>
</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>
</table>
“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)

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
- 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)

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**
- 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)

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**
- 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)

That’s what we will look into in the next sections.
Event-B – Introduction
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 $\text{B}$ 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

systems state evolution over time, triggered by events

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:**

<table>
<thead>
<tr>
<th><strong>Contexts</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ <strong>Carrier sets</strong> (ground types, universes, “urelements”)</td>
</tr>
<tr>
<td>✓ <strong>Constants</strong> (state-independent symbols, rigid symbols)</td>
</tr>
<tr>
<td>✓ <strong>Axioms</strong> (formulas valid by stipulation)</td>
</tr>
<tr>
<td>✓ <strong>Theorems</strong> (formulas proved valid)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Machines</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓ <strong>Context references</strong> (which symbols are available)</td>
</tr>
<tr>
<td>✓ <strong>Variables</strong> (state-dependent symbols, non-rigid symbols, program variables)</td>
</tr>
<tr>
<td>✓ <strong>Invariants</strong> (formulas true in every reachable system state)</td>
</tr>
<tr>
<td>✓ <strong>Events</strong> (state transition descriptions)</td>
</tr>
</tbody>
</table>

(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**:

<table>
<thead>
<tr>
<th>Contexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>- <em>Carrier sets</em> (ground types, universes, “urelements”)</td>
</tr>
<tr>
<td>- <em>Constants</em> (state-independent symbols, rigid symbols)</td>
</tr>
<tr>
<td>- <em>Axioms</em> (formulas valid by stipulation)</td>
</tr>
<tr>
<td>- <em>Theorems</em> (formulas proved valid)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>- <em>Context references</em> (which symbols are available)</td>
</tr>
<tr>
<td>- <em>Variables</em> (state-dependent symbols, non-rigid symbols, program variables)</td>
</tr>
<tr>
<td>- <em>Invariants</em> (formulas true in every reachable system state)</td>
</tr>
<tr>
<td>- <em>Events</em> (state transition descriptions)</td>
</tr>
</tbody>
</table>

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

→ 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.
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.
CONTEXT ExamCtxt
Exam Context

CONTEXT ExamCtxt

SETS

STUDENT // see requirement R1
SUBJECT
Exam Context

CONTEXT ExamCtxt

SETS

STUDENT // see requirement R1
SUBJECT

CONSTANTS

maths physics 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
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

siblings ⊆ STUDENT × STUDENT // function type
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
siblings ⊆ STUDENT × STUDENT // function type
∀s · s ∈ STUDENT ⇒ (s ↦ s) ∉ siblings // irreflexive
// ...
MACHINE ExamAbstract
MACHINE ExamAbstract
SEES ExamCtxt
MACHINE ExamAbstract
SEES ExamCtxt

VARIABLES
  passed  failed
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
MACHINE ExamAbstract
SEES ExamCtxt

VARIABLES

passed failed

INVARIANTS

\[ \text{passed} \subseteq \text{STUDENT} \quad \text{failed} \subseteq \text{STUDENT} \]
\[ \text{passed} \cap \text{failed} = \emptyset \] // R3

EVENTS

INITIALISATION \( \cong \ldots \) // R2
ATTEMPT EXAM \( \cong \ldots \) // R2
PASS EXAM \( \cong \ldots \) // R3
FAIL EXAM \( \cong \ldots \) // R3
MACHINE ExamAbstract
VARIABLES passed failed ...

EVENTS
INITIALISATION \( \hat{=} \)
\[
\begin{align*}
\text{failed} & \colon= \emptyset \\
\text{passed} & \colon= \emptyset
\end{align*}
\]
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 ≜
  \textit{failed} := ∅
  \textit{passed} := ∅

\textsc{PassExam} ≜
  \text{ANY } s \text{ grade}
  \text{WHERE } s \in \textit{STUDENT} \land \text{grade} \leq 4
  \text{THEN } \textit{passed} := \textit{passed} \cup \{s\}

\textsc{FailExam} ≜
  \text{ANY } s \text{ grade}
  \text{WHERE } s \in \textit{STUDENT} \land \text{grade} > 4
  \text{THEN } \textit{failed} := \textit{failed} \cup \{s\}
MACHINE ExamAbstract
VARIABLES passed failed
INVARIANTS passed $\cap$ failed = $\emptyset$ ... 

EVENTS
PASS_exam $\triangleq$
   ANY s grade
   WHERE s $\in$ STUDENT $\land$ grade $\leq$ 4
   THEN passed := passed $\cup$ {s}

FAIL_exam $\triangleq$
   ANY s grade
   WHERE s $\in$ STUDENT $\land$ grade $>$ 4
   THEN failed := failed $\cup$ {s}
MACHINE ExamAbstract
VARIABLES passed failed
INVARINTS passed ∩ failed = ∅  ... 

EVENTS
PASS\_EXAM \triangleq 
  ANY s grade
  WHERE s ∈ STUDENT \ (failed ∪ passed) ∧ grade ≤ 4
  THEN passed := passed ∪ \{s\}

FAIL\_EXAM \triangleq 
  ANY s grade
  WHERE s ∈ STUDENT \ (failed ∪ passed) ∧ grade > 4
  THEN failed := failed ∪ \{s\}
Underspecified model

EVENTS

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

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

\[
\text{ATTEMPT}\, \text{EXAM} \equiv \\
\quad \text{ANY } s \text{ grade WHERE } grade \in \mathbb{N} \land s \in \ldots \\
\text{THEN } skip
\]
Underspecified model

EVENTS

PASS\text{EXAM} \triangleq
\begin{align*}
\text{ANY } s\text{ grade WHERE } grade &\leq 4 \land s \in \ldots \\
\text{THEN } passed &\triangleq passed \cup \{s\}
\end{align*}

FAIL\text{EXAM} \triangleq
\begin{align*}
\text{ANY } s\text{ grade WHERE } grade &> 4 \land s \in \ldots \\
\text{THEN } failed &\triangleq failed \cup \{s\}
\end{align*}

ATTEMPT\text{EXAM} \triangleq
\begin{align*}
\text{ANY } s\text{ grade WHERE } grade &\in \mathbb{N} \land s \in \ldots \\
\text{THEN } skip
\end{align*}

Additional requirement

R4 Any student may attempt the exam three times and ultimately fails if the fourth attempt is unsuccessful.
MACHINE $RefinedExams$ REFINES $ExamsAbstract$
MACHINE *RefinedExams* REFINES *ExamsAbstract*

**VARIABLES** passed attempts
MACHINE RefinedExams REFINES ExamsAbstract
VARIABLES passed attempts
INVARIANTS
  attempts ∈ STUDENT → ℕ // typing for attempts
  failed = \{ s · attempts(s) = 4\} // 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 \( \triangleright \) 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

\[ \text{attempts} \in \text{STUDENT} \rightarrow \mathbb{N} \] \hfill // typing for `attempts`
\[ \text{failed} = \{ s \cdot \text{attempts}(s) = 4 \} \] \hfill // coupling invariant

EVENTS

INITIALISATION \( \sqsupseteq \) REFINES INITIALISATION

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

\text{EXAMULTIMATEFAIL} \sqsupseteq \text{REFINES EXAMFAIL} \ldots
\text{EXAMMISSED} \sqsupseteq \text{REFINES EXAMATTEMPT} \ldots
\text{EXAMPASSED} \sqsupseteq \text{REFINES EXAMPASSED} \ldots
Refinement Exams (2)

... EVENTS

\[
\text{EXAM\_ULTIMATE\_FAIL} \supseteq \text{REFINES} \ \text{EXAM\_FAIL} \\
\text{ANY } s \ grade \\
\text{WHERE } ... \land grade > 4 \land \text{attempts}(s) = 3 \\
\text{THEN} \\
\text{attempts}(s) := \text{attempts}(s) + 1
\]

\[
\text{EXAM\_MISSED} \supseteq \text{REFINES} \ \text{EXAM\_ATTEMPT} \\
\text{ANY } s \ grade \\
\text{WHERE } ... \land grade > 4 \land \text{attempts}(s) < 3 \\
\text{THEN} \\
\text{attempts}(s) := \text{attempts}(s) + 1
\]

...
Refinement Exams (3)

This refinement takes now also R4 into account.

Refinement preserves invariants

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

⇒ 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

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

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 \( \mathbb{Z} \) 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.  
   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 \( \mathbb{Z} \) 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 \( \mathbb{Z} \) 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 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
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

**Russell’s paradox**

Assume that the expression \( \{ s \mid \phi \} \) for any formula \( \phi \) denotes a set. Let \( R := \{ s \mid s \not\in s \} \).
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 \( R := \{ s \mid s \not\in s \} \).

One observes: \( R \in R \iff R \not\in R \) \( \Downarrow \)
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 \( R := \{ s \mid s \not\in s \} \). Not allowed with types.
One observes: \( R \in R \iff R \not\in R \)

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

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

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

- **carrier sets** $C : \mathcal{P}(C)$
  
  example: $STUDENT : \mathcal{P}(STUDENT)$

- **powerset** $\mathcal{P}(\cdot) : \mathcal{P}(S) \to \mathcal{P}(\mathcal{P}(S))$
  
  example: $\mathcal{P}(\{1, 2\}) = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\} : \mathcal{P}(\mathbb{Z})$

- **product** $\times : \mathcal{P}(S) \times \mathcal{P}(T) \to \mathcal{P}(S \times T)$
  
  example: $BOOL \times \{1\} = \{\{\text{true}, 1\}, \{\text{false}, 1\}\} : \mathcal{P}(BOOL \times \mathbb{Z})$

- **set comprehension** $\{x \cdot \varphi | e\}$
  
  formula $\varphi$, term $e : T$, result of type $\mathcal{P}(T)$
  
  example: $\{x \cdot x \geq 2 | x \cdot x\} = \{4, 9, 16, \ldots\}$
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 \((E_1 \mapsto E_2) : \tau(E_1) \times \tau(E_2)\)
- Domain of a relation \( \text{dom}(R) \)
  \( \text{dom}(R) = \{x, y : (x \mapsto y) \in R \mid x\} \)
  example: \( \text{dom}(E_1 \times E_2) = E_1 \)

Klebanov, Ulbrich – Applications of Formal Verification
Relations

- Relations are sets of pairs (tuples).

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

- Pairs \( (E_1 \mapsto E_2) : \tau(E_1) \times \tau(E_2) \)

- Domain of a relation \( \text{dom}(R) \)
  \[ \text{dom}(R) = \{ x, y \cdot (x \mapsto y) \in R \mid x \} \]
  example: \( \text{dom}(E_1 \times E_2) = E_1 \)

- Range of a relation \( \text{ran}(R) \)
  \[ \text{ran}(R) = \{ x, y \cdot (x \mapsto y) \in R \mid y \} \]
  example: \( \text{ran}(E_1 \times E_2) = E_2 \)
Relations

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

- All relations $E_1 \leftrightarrow E_2$
Kinds of relations

- All relations $E_1 \leftrightarrow E_2$

- All surjections $E_1 \leftrightarrow E_2 \quad (\text{ran}(R) = E_2)$
Kinds of relations

- All relations $E_1 \leftrightarrow E_2$

- All surjections $E_1 \leftrightarrow E_2 \quad \text{(ran}(R) = E_2)$

- All total relations $E_1 \leftrightarrow E_2 \quad \text{(dom}(R) = E_1)$
Kinds of relations

- All relations \( E_1 \leftrightarrow E_2 \)

- All surjections \( E_1 \leftrightarrow E_2 \) \( (\text{ran}(R) = E_2) \)

- All total relations \( E_1 \leftrightarrow E_2 \) \( (\text{dom}(R) = E_1) \)

- All total surjections \( E_1 \leftrightarrow E_2 \)
Functional relations

Observation

Every function \( f \in A \to B \) can be understood as the relation
\[
\{ x \cdot x \in A \mid x \mapsto f(x) \} \in A \leftrightarrow B
\]

- Partial functions \( E_1 \equiv E_2 \subseteq E_1 \leftrightarrow E_2 \)
  \[(\forall x, y, z \cdot x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z) \quad (\ast)\]
Functional relations

Observation

Every function \( f \in A \rightarrow B \) can be understood as the relation

\[
\{ x \cdot x \in A \mid x \mapsto f(x) \} \in A \leftrightarrow B
\]

- Partial functions \( E_1 \leftrightarrow E_2 \subseteq E_1 \leftrightarrow E_2 \)
  \( (\forall x, y, z \cdot x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z) (\ast) \)

- Total functions \( E_1 \rightarrow E_2 \)
  \( E_1 \rightarrow E_2 = (E_1 \leftrightarrow E_2) \cap (E_1 \leftrightarrow E_2) \)
  (both partial function and total relation)
Functional relations

Observation

Every function \( f \in A \rightarrow B \) can be understood as the relation
\[
\{ x \cdot x \in A \mid x \mapsto f(x) \} \in A \leftrightarrow B
\]

- Partial functions \( E_1 \mapsto E_2 \subseteq E_1 \leftrightarrow E_2 \)
  \((\forall x, y, z \cdot x \mapsto y \in R \land x \mapsto z \in R \Rightarrow y = z)\)

- Total functions \( E_1 \rightarrow E_2 \)
  \(E_1 \rightarrow E_2 = (E_1 \mapsto E_2) \cap (E_1 \leftrightarrow E_2)\)
  (both partial function and total relation)

- Injections \( E_1 \leftrightarrow E_2 \)
  \((\ast) \land (\forall x, y, z \cdot x \mapsto z \in R \land y \mapsto z \in R \Rightarrow x = y)\)
Functional relations (2)

Intersection of relation classes give new classes:

- Total injections $E_1 \leftrightarrow E_2 = (E_1 \to E_2) \cap (E_1 \leftrightarrow E_2)$
- Partial surjections $E_1 \leftrightarrow E_2 = (E_1 \to E_2) \cap (E_1 \leftrightarrow E_2)$
- Total surjections $E_1 \to E_2 = (E_1 \to E_2) \cap (E_1 \to E_2)$
- Bijections $E_1 \leftrightarrow E_2 = (E_1 \to E_2) \cap (E_1 \leftrightarrow E_2)$
Example: File system

CONTEXT FileSystemCtx
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT

CONSTANTS files, dirs, root

AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT, root ∈ dirs, files ∩ dirs = ∅
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT,
  root ∈ dirs, files ∩ dirs = ∅
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT,
root ∈ dirs, files ∩ dirs = ∅

MACHINE FileSystem SEES FileSystemCtx
VARIABLES tree, parent
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT,
      root ∈ dirs, files ∩ dirs = ∅

MACHINE FileSystem SEES FileSystemCtx
VARIABLES tree, parent
INVARIANTS
      tree ∈ dirs ↔ (files ∪ dirs)
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files \subseteq OBJECT, dirs \subseteq OBJECT,
     root \in dirs, files \cap dirs = \emptyset

MACHINE FileSystem SEES FileSystemCtx
VARIABLES tree, parent
INvariants
    tree \in dirs \iff (files \cup dirs)
        \// most directories (but root) have a parent directory :
    parent \in dirs \Rightarrow dirs
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT,
    root ∈ dirs, files ∩ dirs = ∅

MACHINE FileSystem SEES FileSystemCtx
VARIABLES tree, parent
INVARIANTS
    tree ∈ dirs ↔ (files ∪ dirs)
    // most directories (but root) have a parent directory :
    parent ∈ dirs → dirs
    // more precise
    parent ∈ (dirs \ {root}) → dirs
Relational operations

- Relational application \( \cdot[\cdot] : \mathcal{P}(S \times T) \times \mathcal{P}(S) \rightarrow \mathcal{P}(T) \)

\[
R[A] = \{ x, y \cdot x \mapsto y \in R \land x \in A \mid y \}
\]
Relational operations

- Relational application \( \cdot[\cdot] : \mathbb{P}(S \times T) \times \mathbb{P}(S) \to \mathbb{P}(T) \)

\[ R[A] = \{ x, y \cdot x \mapsto y \in R \land x \in A \mid y \} \]
Relational operations

- Relational application \( \cdot [\cdot] : \mathcal{P}(S \times T) \times \mathcal{P}(S) \rightarrow \mathcal{P}(T) \)

\[
R[A] = \{ x, y \cdot x \mapsto y \in R \wedge x \in A \mid y \}
\]

Diagram:

- \( B \)
- \( \text{dom} \)
- \( \text{ran} \)
- \( R \)
- \( R[B] = \emptyset \)
Relational operations

- Relational application \( \cdot [\cdot] : \mathcal{P}(S \times T) \times \mathcal{P}(S) \rightarrow \mathcal{P}(T) \)
  \[
  R[A] = \{ x, y \cdot x \mapsto y \in R \land x \in A \mid y \}
  \]

- Functional application \( \cdot (\cdot) : \mathcal{P}(S \times T) \times S \rightarrow T \)
  \[
  x = f(e) \iff e \mapsto x \in f \quad \{ f(e) \} = f[\{ e \}]
  \]
Relational operations

- **Relational application** [· : \( \mathbb{P}(S \times T) \times \mathbb{P}(S) \rightarrow \mathbb{P}(T) \)]
  
  \[ R[A] = \{ x, y \cdot x \mapsto y \in R \land x \in A \mid y \} \]

- **Functional application** [·(·) : \( \mathbb{P}(S \times T) \times S \rightarrow T \)]

  \[ x = f(e) \iff e \mapsto x \in f \quad \{f(e)\} = f[{\{e}\}] \]

Problem: What if \( f[{\{e}\}] \) is not a one-element set?
Relational operations

- Relational application \( \cdot [\cdot] : \mathcal{P}(S \times T) \times \mathcal{P}(S) \to \mathcal{P}(T) \)

\[
R[A] = \{ x, y : x \mapsto y \in R \land x \in A \mid y \}\]

- Functional application \( \cdot (\cdot) : \mathcal{P}(S \times T) \times S \to T \)

\[
x = f(e) \iff e \mapsto x \in f \quad \{ f(e) \} = f[\{ e \}]
\]

Problem: What if \( f[\{ e \}] \) is not a one-element set?
Solution: Well-definedness needs to be proved

1. \( f \in S \mapsto T \) (not an arbitrary relation in \( S \leftrightarrow T \))
2. \( e \in \text{dom}(f) \)

everytime a functional application is used.
Restrictions

Concept

Limit the domain or range of a relation to a subset.

\[
\text{dom} \quad \rightarrow \quad \text{ran}
\]
Restrictions

Concept

Limit the domain or range of a relation to a subset.
Restrictions

Concept

Limit the domain or range of a relation to a subset.

\[ A \triangleleft R \]

\[ \text{dom} \quad \text{ran} \]
Restrictions

Concept

Limit the domain or range of a relation to a subset.

\[
A \triangleleft R := \{ x, y : x \rightarrow y \in R \land x \in A \mid x \rightarrow y \} \subseteq R
\]

\[
A \triangleleft R := \{ x, y : x \rightarrow y \in R \land x \not\in A \mid x \rightarrow y \} \subseteq R
\]

\[
R \triangleright B := \{ x, y : x \rightarrow y \in R \land y \in B \mid x \rightarrow y \} \subseteq R
\]

\[
R \triangleright B := \{ x, y : x \rightarrow y \in R \land y \not\in B \mid x \rightarrow y \} \subseteq R
\]
Restrictions

Concept

Limit the domain or range of a relation to a subset.

\[ A \bowtie R := \{ x, y \mid x \in A \land x \mapsto y \in R \} \subseteq R \]

\[ A \bowtie R := \{ x, y \mid x \in A \land x \mapsto y \not\in R \} \subseteq R \]

\[ R \triangleright B := \{ x, y \mid x \mapsto y \in R \land y \in B \} \subseteq R \]

\[ R \triangleright B := \{ x, y \mid x \mapsto y \in R \land y \not\in B \} \subseteq R \]

*Relational application:* \( R[A] = \text{ran}(A \bowtie R) \)
Override

\[ R \leftarrow S := ((\text{dom } S) \leftarrow R) \cup S \]

\[ x \mapsto y \in R \leftarrow S \iff \begin{cases} 
  x \mapsto y \in S & \text{if } x \in \text{dom}(S) \\
  x \mapsto y \in R & \text{if } x \notin \text{dom}(S)
\end{cases} \]

- “Clear” \( \text{dom}(S) \) in \( R \) and “replace” by \( S \).
Override

\[ R \leftarrow S := ((\text{dom } S) \leftarrow R) \cup S \]

\[ x \mapsto y \in R \leftarrow S \iff \begin{cases} x \mapsto y \in S & \text{if } x \in \text{dom}(S) \\ x \mapsto y \in R & \text{if } x \notin \text{dom}(S) \end{cases} \]

- "Clear" \( \text{dom}(S) \) in \( R \) and "replace" by \( S \).
- Special case: \( f \in A \rightarrow B, g \in A \rightarrow B \) implies \( f \leftarrow g \in A \rightarrow B \)
Override

\[ R \leftarrow S := ((\text{dom } S) \leftarrow R) \cup S \]

\[ x \mapsto y \in R \leftarrow S \iff \begin{cases} x \mapsto y \in S & \text{if } x \in \text{dom}(S) \\ x \mapsto y \in R & \text{if } x \notin \text{dom}(S) \end{cases} \]

- “Clear” \( \text{dom}(S) \) in \( R \) and “replace” by \( S \).
- Special case: \( f \in A \rightarrow B, g \in A \rightarrow B \) implies \( f \leftarrow g \in A \rightarrow B \)
- \( f \leftarrow \{ x \mapsto y \} \) updates function \( f \) in one place \( x \)
Override

\[ R \triangleleft S := ((\text{dom } S) \triangleleft R) \cup S \]

\[ x \mapsto y \in R \triangleleft S \iff \begin{cases} x \mapsto y \in S & \text{if } x \in \text{dom}(S) \\ x \mapsto y \in R & \text{if } x \not\in \text{dom}(S) \end{cases} \]

- “Clear” \( \text{dom}(S) \) in \( R \) and “replace” by \( S \).
- Special case: \( f \in A \to B, g \in A \to B \) implies \( f \triangleleft g \in A \to B \)
- \( f \triangleleft \{ x \mapsto y \} \) updates function \( f \) in one place \( x \)
- Caution: \( \triangleleft \) and \( \triangleleft \) are different symbols
Override

\[ R \leftarrow S := ((\text{dom } S) \leftarrow R) \cup S \]

\[ x \mapsto y \in R \leftarrow S \iff \begin{cases} x \mapsto y \in S & \text{if } x \in \text{dom}(S) \\ x \mapsto y \in R & \text{if } x \notin \text{dom}(S) \end{cases} \]

- “Clear” \( \text{dom}(S) \) in \( R \) and “replace” by \( S \).
- Special case: \( f \in A \rightarrow B, g \in A \rightarrow B \) implies \( f \leftarrow g \in A \rightarrow B \)
- \( f \leftarrow \{x \mapsto y\} \) updates function \( f \) in one place \( x \)

- Caution: \( \leftarrow \) and \( \leftarrow \) are different symbols
- Syntax sometimes \( \oplus \) instead of \( \leftarrow \)
Override

\[ R \leftarrow S := ((\text{dom } S) \leftarrow R) \cup S \]

\[ x \mapsto y \in R \leftarrow S \iff \begin{cases} x \mapsto y \in S & \text{if } x \in \text{dom}(S) \\ x \mapsto y \in R & \text{if } x \not\in \text{dom}(S) \end{cases} \]

- “Clear” \( \text{dom}(S) \) in \( R \) and “replace” by \( S \).
- Special case: \( f \in A \rightarrow B, g \in A \rightarrow B \) implies \( f \leftarrow g \in A \rightarrow B \)
- \( f \leftarrow \{ x \mapsto y \} \) updates function \( f \) in one place \( x \)

- Caution: \( \leftarrow \) and \( \leftarrow \) are different symbols
- Syntax sometimes \( \oplus \) instead of \( \leftarrow \)
- Compare \textit{Updates} in Dynamic Logic for KeY.
Forward composition

\[ x \mapsto y \in R ; S \iff \exists z \cdot x \mapsto z \in R \land z \mapsto y \in S \]

\(x \mapsto y\) is in the composition \(R ; S\) if there is a transmitting element \(z\) with both \(x \mapsto z \in R\) and \(z \mapsto y \in S\).
Forward composition

\[ x \mapsto y \in R ; S \iff \exists z \cdot x \mapsto z \in R \land z \mapsto y \in S \]

\( x \mapsto y \) is in the composition \( R ; S \) if there is a transmitting element \( z \) with both \( x \mapsto z \in R \) and \( z \mapsto y \in S \).
Forward composition

\[ x \mapsto y \in R \land S \iff \exists z \cdot x \mapsto z \in R \land z \mapsto y \in S \]

\( x \mapsto y \) is in the composition \( R ; S \) if there is a transmitting element \( z \) with both \( x \mapsto z \in R \) and \( z \mapsto y \in S \).

(There is also backward composition \( R \circ S = S ; R \))
Example: File system

CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT,
      root ∈ dirs, files ∩ dirs = ∅

MACHINE FileSystem SEES FileSystemCtx
VARIABLES tree, depth
INVARIABLES
      tree ∈ dirs ↔ (files ∪ dirs) ∧ depth ∈ dirs → ℕ ∧
Example: File system

```
CONTEXT FileSystemCtx
SETS OBJECT
CONSTANTS files, dirs, root
AXIOMS files ⊆ OBJECT, dirs ⊆ OBJECT,
      root ∈ dirs, files ∩ dirs = Ø

MACHINE FileSystem SEES FileSystemCtx
VARIABLES tree, depth
INVARIANTS
  tree ∈ dirs ↔ (files ∪ dirs) ∧ depth ∈ dirs → ℕ ∧
  ∀d · ((depth(d) > 0 ⇒ depth[tree[{d}]] = {depth(d) − 1})
     ∧ (depth(d) = 0 ⇒ {d} ◄ tree ⊢ files = Ø))
```
The symbols in context can be used in \( \text{inv} \) even if not mentioned explicitly.
There is one more construct (WITH) that we omit here.
**Actions** (Generalised Substitutions)

**Deterministic actions**
- “Assignment” $x := t$
- Variable $x$ and term $t$ must have same type ($\tau(t) = \tau(x)$)
- After event, $x$ has value of expression $t$
Deterministic actions

- “Assignment” $x := t$
- Variable $x$ and term $t$ must have same type ($\tau(t) = \tau(x)$)
- After event, $x$ has value of expression $t$
Deterministic actions

- "Assignment" $x := t$
- Variable $x$ and term $t$ must have same type ($\tau(t) = \tau(x)$)
- After event, $x$ has value of expression $t$
Actions (Generalised Substitutions)

Deterministic actions

- “Assignment” $x := t$
- Variable $x$ and term $t$ must have same type ($\tau(t) = \tau(x)$)
- After event, $x$ has value of expression $t$
Actions (Generalised Substitutions)

Deterministic actions

- “Assignment” $x := t$
- Variable $x$ and term $t$ must have same type ($\tau(t) = \tau(x)$)
- After event, $x$ has value of expression $t$

Example:

```
THEN
  x := y
  y := x
END  // swaps values of variables $x$, $y$.
```

Unmentioned variable $z$ does not change.

Remember the updates in KeY: $\{x := y || y := x\}$ has same effects.
Actions (Generalised Substitutions)

<table>
<thead>
<tr>
<th>Nondeterministic actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x :</td>
</tr>
</tbody>
</table>

- Actions can have more than one resolution
- $\varphi$ is called the before-after-predicate (BAP)
- Variables without tick: before-state
- Variables with tick: after-state.
Actions (Generalised Substitutions)

Nondeterministic actions

\[ x : \mid \varphi \] means “choose \( x \) such that \( \varphi \)”

- Actions can have more than one resolution
- \( \varphi \) is called the before-after-predicate (BAP)
- Variables without tick: before-state
- Variables with tick: after-state.
Actions (Generalised Substitutions)

Nondeterministic actions

\[ x : \varphi \] means “choose \( x \) such that \( \varphi \)”

- Actions can have more than one resolution
- \( \varphi \) is called the before-after-predicate (BAP)
- variables without tick: before-state
- variables with tick: after-state.

Example: \( x, y : x' = y' \land y' > y \)

After the action \( x \) and \( y \) are equal and \( y \) is strictly greater than before the action.
Nondeterministic actions

\[ x : \varphi \] means “choose \( x \) such that \( \varphi \)”

- Actions can have more than one resolution
- \( \varphi \) is called the before-after-predicate (BAP)
- variables without tick: before-state
- variables with tick: after-state.

Example:

\[ x, y : \varphi \quad \text{where} \quad x' = y' \land y' > y \]

After the action \( x \) and \( y \) are equal and \( y \) is strictly greater than before the action.
**Actions (Generalised Substitutions)**

### Nondeterministic actions

\[ x :| \varphi \Rightarrow \text{“choose } x \text{ such that } \varphi \text{”} \]

- Actions can have more than one resolution
- \( \varphi \) is called the before-after-predicate (BAP)
- Variables without tick: before-state
- Variables with tick: after-state.
Nondeterministic actions

$x : \varphi$ means “choose $x$ such that $\varphi$”

- Actions can have more than one resolution
- $\varphi$ is called the before-after-predicate (BAP)
- Variables without tick: before-state
- Variables with tick: after-state.

**Example:**

$$x, y : x' = y' \land y' > y$$

*After* the action $x$ and $y$ are equal and $y$ is strictly greater than before the action.
Actions (Generalised Substitutions)

Normal form

Every action can be defined as a before-after-predicate

\[ bap(\text{vars}, \text{vars}', \text{prms}) \]

with

1. \text{vars} the machines variables before the action
2. \text{vars}' the machine variables after the action
3. \text{prms} the parameters of the event

- \( x := t \) is short for \( x :| x' = t \)
- \( x :\in S \) is short for \( x :| x' \in S \)
Initialisation

- Values of the machine in the beginning?
Initialisation

- Values of the machine in the beginning?
- Initial values defined by the special event INITIALISATION.
Initialisation

- Values of the machine in the beginning?
- Initial values defined by the special event INITIALISATION.
- before-after-predicate $bap_{init}$ and guard $grd_{init}$ must not refer to $vars$, there is no “before-state”.
Initialisation

- Values of the machine in the beginning?

- Initial values defined by the special event INITIALISATION.

- before-after-predicate $bap_{init}$ and guard $grd_{init}$ must not refer to $vars$, there is no “before-state”.

- After the first state, only normal events trigger.
Machine Semantics

Machine variables \( \text{vars} := v_1, \ldots, v_k \) with types \( \overline{T} = T_1 \times \ldots \times T_k \).

A state \( \sigma \in \overline{T} \) is a vector, variable assignment.

A trace is a sequence of states \( \sigma_0, \sigma_1, \ldots \) such that

- first state \( \sigma_0 \) is result of the initialisation event
- every state \( \sigma_i \) results from an event which operates on \( \sigma_{i-1} \) (for every \( i > 0 \)).

The semantics of a machine \( M \) is the set of all traces possible for \( M \).
Event Parameters

Sources for indeterminism

- indeterministic choices in bap’s (cf. $: \in, :|$)
- event parameters

Event parameter may model:

- content of messages passed around
- indeterministic user input
- unpredictable environment actions
- a number, amount of data to operate with
- ...

Technically event parameters can be removed and replaced by existential quantifiers.
Semantics (more formally)

State space: $\overline{T} = T_1 \times \ldots \times T_k$
Semantics (more formally)

State space: $\overline{T} = T_1 \times \ldots \times T_k$

Trace: $t \in \mathbb{N} \rightarrow \overline{T}$
Semantics (more formally)

State space: $\overline{T} = T_1 \times \ldots \times T_k$

Trace: $t \in \mathbb{N} \rightarrow \overline{T}$
with

- $\exists \text{prms}_{init} \cdot \text{grd}_{init}(\text{prms}_{init}) \land \text{bap}_{init}(t(0), \text{prms}_{init})$
Semantics (more formally)

State space: $\overline{T} = T_1 \times \ldots \times T_k$

Trace: $t \in \mathbb{N} \rightarrow \overline{T}$
with

- $\exists \text{prms}_{\text{init}} \cdot \text{grd}_{\text{init}}(\text{prms}_{\text{init}}) \land \text{bap}_{\text{init}}(t(0), \text{prms}_{\text{init}})$
- For $n \in \mathbb{N}_1$, there is $e \in \text{EVENTS}$ such that $\exists \text{prms}_e \cdot \text{grd}_e(t(i - 1), \text{prms}_e) \land \text{bap}_e(t(i - 1), t(i), \text{prms}_e)$
Semantics (more formally)

State space: \( \overline{T} = T_1 \times \ldots \times T_k \)

Trace: \( t \in \mathbb{N} \rightarrow \overline{T} \)
with

- \( \exists prms_{init} \cdot \text{grd}_{init}(prms_{init}) \land \text{bap}_{init}(t(0), prms_{init}) \)
- For \( n \in \mathbb{N}_1 \), there is \( e \in EVENTS \) such that
  \( \exists prms_e \cdot \text{grd}_e(t(i - 1), prms_e) \land \text{bap}_e(t(i - 1), t(i), prms_e) \)
Semantics (more formally)

State space: $\overline{T} = T_1 \times \ldots \times T_k$

Trace: $t \in \mathbb{N} \rightarrow \overline{T}$

with

- $\exists \text{prms}_{init} \cdot \text{grd}_{init}(\text{prms}_{init}) \land \text{bap}_{init}(t(0), \text{prms}_{init})$

- For $n \in \mathbb{N}_1$, there is $e \in \text{EVENTS}$ such that
  $\exists \text{prms}_e \cdot \text{grd}_e(t(i - 1), \text{prms}_e) \land \text{bap}_e(t(i - 1), t(i), \text{prms}_e)$

Partial, finite trace trace: $t \in 0..n \rightarrow \overline{T}$
Semantics (more formally)

State space: \( \overline{T} = T_1 \times \ldots \times T_k \)

Trace: \( t \in \mathbb{N} \rightarrow \overline{T} \) with
  
  \[ \exists \text{prms}_{\text{init}} \cdot \text{grd}_{\text{init}}(\text{prms}_{\text{init}}) \land \text{bap}_{\text{init}}(t(0), \text{prms}_{\text{init}}) \]

  For \( n \in \mathbb{N}_1 \), there is \( e \in \text{EVENTS} \) such that
  
  \[ \exists \text{prms}_e \cdot \text{grd}_e(t(i - 1), \text{prms}_e) \land \text{bap}_e(t(i - 1), t(i), \text{prms}_e) \]

Partial, finite trace trace: \( t \in 0..n \rightarrow \overline{T} \)

Deadlock: no event \( e \) can be triggered, i.e.

\[ \forall \text{prms}_e \cdot \neg \text{grd}_e(t(n), \text{prms}_e) \] for all events \( e \).
**SAFETY**: Do all states reachable by $M$ satisfy $inv$?

The red trace violates the invariant in two states.
To show that $inv(\text{vars})$ is an invariant for machine $M$, one proves for every event:

<table>
<thead>
<tr>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guards of the event</td>
</tr>
<tr>
<td>Before-after-predicate of the event</td>
</tr>
</tbody>
</table>

$\Rightarrow$

modified invariant
Proof Obligation INV

To show that $inv(\text{vars})$ is an invariant for machine $M$, one proves:

1. $\forall prms, vars' \cdot \text{grd}_{\text{init}}(prms) \land \text{bap}_{\text{init}}(vars', prms) \rightarrow inv(vars')$

(Invariant initially valid)
Proof Obligation INV

To show that $inv(\text{vars})$ is an invariant for machine $M$, one proves:

1. $\forall prms, vars'. \quad \text{grd}_{init}(prms) \land bap_{init}(vars', prms) \rightarrow inv(vars')$
   (Invariant initially valid)

2. $\forall prms, vars, vars'. \quad inv(vars) \land \text{grd}_e(vars, prms) \land bap_e(vars, vars', prms) \rightarrow inv(vars')$
   for every event $e$ in $M$.
   (Events preserve invariant)
Proof Obligation INV

To show that \( inv(\text{vars}) \) is an invariant for machine \( M \), one proves:

1. \( \forall \text{prms}, \text{vars}'. \)
   \[ \text{grd}_{init} \land \text{bap}_{init} \rightarrow inv \]
   (Invariant initially valid)

2. \( \forall \text{prms}, \text{vars}, \text{vars}' \cdot \)
   \[ \text{inv} \land \text{grd}_e \land \text{bap}_e \rightarrow inv' \]
   for every event \( e \) in \( M \).
   (Events preserve invariant)
Proof Obligation INV

To show that \( \text{inv}(\text{vars}) \) is an invariant for machine \( M \), one proves:

1. \( \forall \text{prms}, \text{vars}'. \)
   \[ \text{grd}_{\text{init}} \land \text{bap}_{\text{init}} \rightarrow \text{inv} \]
   (Invariant initially valid)

2. \( \forall \text{prms}, \text{vars}, \text{vars}' \cdot \)
   \[ \text{inv} \land \text{grd}_e \land \text{bap}_e \rightarrow \text{inv}' \]
   for every event \( e \) in \( M \).
   (Events preserve invariant)

Note: Proof Obligation INV is a sufficient criterion, but not necessary. Necessary for inductive invariants.
Inductive Invariant

MACHINE \textit{IndInv}
VARIABLES $x$\quad INVARIANTS $x \in \mathbb{Z} \quad x \geq 0$
EVENTS

\begin{align*}
\text{INITIALISATION} \triangleq \\
& x := 2 \\
\text{STEP} \triangleq \\
& x := 2 * (x - 1)
\end{align*}

There is only one trace:

$(2, 2, 2, 2, \ldots)$

invariant is fulfilled.
Inductive Invariant – Won’t prove

Proof obligation INV for event STEP

\[ inv(x) \land grd(x) \land bap(x, x') \rightarrow inv(x') \]

This is not valid! Invariant is not inductive.

Counter-example: \( x = 0, x' = -2 \)

All states

Reachable states
Inductive Invariant – Won’t prove

**Proof obligation INV for event STEP**

\[
\begin{align*}
inv(x) \land \text{grd}(x) \land bap(x, x') & \rightarrow inv(x') \\
x \geq 0 \land x' = 2 \cdot (x - 1) & \rightarrow x' \geq 0
\end{align*}
\]

This is not valid! Invariant is not inductive.

Counter-example:

\[x = 0, \quad x' = -2\]
Inductive Invariant – Won’t prove

Proof obligation INV for event STEP

\[ \begin{align*}
\text{inv}(x) & \land \text{grd}(x) \land \text{bap}(x, x') \rightarrow \text{inv}(x') \\
x \geq 0 & \land x' = 2 \times (x - 1) \rightarrow x' \geq 0
\end{align*} \]

\[ \checkmark \text{This is not valid! Invariant is not inductive.} \checkmark \]

Counter-example: \( x = 0, x' = -2 \)
Inductive Invariant – Won’t prove

Proof obligation $\text{INV}$ for event $\text{STEP}$

\[
\begin{align*}
\text{inv}(x) & \land \text{grd}(x) \land \text{bap}(x, x') \Rightarrow \text{inv}(x') \\
x \geq 0 & \land x' = 2 \times (x - 1) \Rightarrow x' \geq 0 \\
\downarrow & \text{ This is not valid! Invariant is not inductive.} \\
\text{Counter-example: } & x = 0, x' = -2
\end{align*}
\]
Show that every action is feasible if the guard is true:

\[\text{Invariants} \quad \text{Guards of the event} \quad \Rightarrow \quad \exists v' \cdot \text{before-after-predicate}\]
Feasibility Proof Obligation FIS

The action of an event is possible if guard is true.

\[ \forall vars, prms \cdot \text{grd}_e(vars, prms) \rightarrow \exists vars' \cdot \text{bap}(vars, vars', prms) \]

Deterministic action: \( x := t \)

\[ \ldots \text{nothing to show} \]

Indeterministic action: \( x : \in S \)

\[ \ldots \text{show that } S \neq \emptyset \]

Indeterministic action: \( x : | \varphi \)

\[ \ldots \text{show satisfiability of } \varphi \]

Thus impossible evolutions like \( x : | \text{false} \) or \( x : \in \emptyset \) are avoided
Recap:
Deadlock: no event $e$ can be triggered, i.e. 
\[ \forall prms_e \cdot \neg grd_e(t(n), prms_e) \] for all events $e$. 
Deadlock Freedom DLKF

Recap:
Deadlock: no event $e$ can be triggered, i.e.
$\forall prms_e \cdot \neg grd_e(t(n), prms_e)$ for all events $e$.

Proof Obligation
There is always an event that can trigger:

$$\forall vars \cdot inv(vars) \Rightarrow \bigvee_{\text{event } e \in M} \exists prms \cdot grd_e(vars, prms)$$
Recap:
Deadlock: no event $e$ can be triggered, i.e. $\forall prms_e \cdot \neg grd_e(t(n), prms_e)$ for all events $e$.

Proof Obligation
There is always an event that can trigger:

$$\forall vars \cdot inv(vars) \Rightarrow \bigvee_{\text{event } e \in M} \exists prms \cdot grd_e(vars, prms)$$

Again, this is sufficient not necessary.
(The invariant may be too weak to imply deadlock freedom)
Event-B – Refinement
Refinement in Event-B
Refinement in Event-B
Refinement in Event-B

MACHINE Abstract
VARIABLES x
INVARIANTS x ≥ 0
EVENTS INCREASE ≜
   x : | x' ≥ x
Refinement in Event-B

MACHINE Abstract
VARIABLES x
INVARIANTS x \geq 0
EVENTS INCREASE \equiv x :| x' \geq x
Refinement in Event-B

**MACHINE** *Abstract*

**VARIABLES** \( x \)

**INVARIANTS** \( x \geq 0 \)

**EVENTS** *INCREASE* ≜

\[
x : x' \geq x
\]

**MACHINE** *Refined*

**REFINES** *Abstract*

**VARIABLES** \( x \)
Refinement in Event-B

MACHINE Abstract
VARIABLES x
INVARIANTS x ≥ 0
EVENTS INCREASE ≜
  x : | x' ≥ x

MACHINE Refined
REFINES Abstract
VARIABLES x
EVENTS NEXT Val ≜
  REFINES INCREASE
  x := 5 * x^2 + 3 * x
Refinement in Event-B

MACHINE Abstract
VARIABLES $x$
INVARIANTS $x \geq 0$
EVENTS INCREASE $\triangleq$

$$x : | x' \geq x$$

MACHINE Refined
REFINES Abstract
VARIABLES $x$
EVENTS NEXT VAL $\triangleq$
REFINES INCREASE

$$x := 5 \times x^2 + 3 \times x$$
Refining Machines

MACHINE Abstract

SEES Context

VARIABLES $vars_A$

INVARIANTS

$inv_A(vars_A)$

EVENTS

INITIALISATION $\models \ldots$

$EVT_A \models \ldots$

END

MACHINE Refined

REFINES Abstract

SEES Context

VARIABLES $vars_R$

INVARIANTS

$inv_R(vars_A, vars_R)$

EVENTS

INITIALISATION $\models \ldots$

$EVT_R \models$

REFINES EVT_A \ldots

END
Machines as Relations

Every machine $M$ defines:
- a state space $S_M$ spanned by the types of $\text{vars}_M$
- the initialisation $I_M \subseteq S_M$
- the transition relations $E_{M;\text{evt}} \in S_M \leftrightarrow S_M$ (for event $\text{evt}$)

**Details**

$$S_M = \tau(v_1) \times \ldots \times \tau(v_k) \quad \text{(with } \text{vars}_M = v_1, \ldots, v_k)$$
Machines as Relations

Every machine $M$ defines:

- a state space $S_M$ spanned by the types of $\text{vars}_M$
- the initialisation $I_M \subseteq S_M$
- the transition relations $E_{M;\text{evt}} \in S_M \leftrightarrow S_M$ (for event $\text{evt}$)

**Details**

\[
S_M = \tau(v_1) \times \ldots \times \tau(v_k) \quad \text{(with } \text{vars}_M = v_1, \ldots, v_k) \\
I_M(p) = \{ s \in S_M \mid \text{grd}_{\text{init}}(p) \land \text{bap}_{\text{init}}(s', p) \} 
\]
Machines as Relations

Every machine $M$ defines:
- a state space $S_M$ spanned by the types of $\text{vars}_M$
- the initialisation $I_M \subseteq S_M$
- the transition relations $E_{M;\text{evt}} \in S_M \leftrightarrow S_M$ (for event $\text{evt}$)

Details

\[
S_M = \tau(v_1) \times \ldots \times \tau(v_k) \quad \text{(with } \text{vars}_M = v_1, \ldots, v_k) \\
I_M(p) = \{ s \in S_M \mid \text{grd}_{\text{init}}(p) \land \text{bap}_{\text{init}}(s', p) \} \\
I_M = \bigcup_{p} I_M(p)
\]
Machines as Relations

Every machine $M$ defines:

- a state space $S_M$ spanned by the types of $\text{vars}_M$
- the initialisation $I_M \subseteq S_M$
- the transition relations $E_{M;\text{evt}} \in S_M \leftrightarrow S_M$ (for event $\text{evt}$)

Details

$$S_M = \tau(v_1) \times \ldots \times \tau(v_k) \quad (\text{with } \text{vars}_M = v_1, \ldots, v_k)$$

$$I_M(p) = \{ s \in S_M \mid \text{grd}_{\text{init}}(p) \land \text{bap}_{\text{init}}(s', p) \}$$

$$I_M = \bigcup_p I_M(p)$$

$$E_{M;\text{evt}}(p) = \{(s \mapsto s') \mid \text{grd}_{\text{evt}}(s, p) \land \text{bap}_{\text{evt}}(s, s', p) \}$$
Machines as Relations

Every machine $M$ defines:
- a state space $S_M$ spanned by the types of $\text{vars}_M$
- the initialisation $I_M \subseteq S_M$
- the transition relations $E_{M;\text{evt}} \in S_M \leftrightarrow S_M$ (for event $\text{evt}$)

Details

$$S_M = \tau(v_1) \times \ldots \times \tau(v_k) \quad \text{(with } \text{vars}_M = v_1, \ldots, v_k)$$

$$I_M(p) = \{ s \in S_M \mid \text{grd}_{\text{init}}(p) \land \text{bap}_{\text{init}}(s', p) \}$$

$$I_M = \bigcup_{p} I_M(p)$$

$$E_{M;\text{evt}}(p) = \{ (s \mapsto s') \mid \text{grd}_{\text{evt}}(s, p) \land \text{bap}_{\text{evt}}(s, s', p) \}$$

$$E_{M;\text{evt}} = \bigcup_{p} E_{M;\text{evt}}(p)$$
Every trace of the refined machine \( R \) is a trace of the abstract machine \( A \).

**Definition: Simple Refinement**

Let \( R, A \) be two machines with the same state space \( S \). \( R \) is called a refinement of \( A \) if

1. \( I_R \subseteq I_A \) and
Simple Refinement – Definition

Every trace of the refined machine \( R \) is a trace of the abstract machine \( A \).

**Definition: Simple Refinement**

Let \( R, A \) be two machines with the same state space \( S \). \( R \) is called a refinement of \( A \) if

1. \( I_R \subseteq I_A \) and
2. \( E_{R;\text{evt}_R} \subseteq E_{A;\text{evt}_A} \) for each event.
Every trace of the refined machine $R$ is a trace of the abstract machine $A$.

**Definition: Simple Refinement**

Let $R, A$ be two machines with the same state space $S$. $R$ is called a refinement of $A$ if

1. $I_R \subseteq I_A$ and
2. $E_{R;evt_R} \subseteq E_{A;evt_A}$ for each event

($evt_R$ is the event in $R$ that refines event $evt_A$ from $A$)
Loss of behaviour

Why is this problematic?

MACHINE A ...
EVENT emergencyStop \equiv
WHERE true THEN heavyMachine := stop
END

refined by

MACHINE R ...
EVENT emergencyStop \equiv \text{REFINES} emergencyStop
WHERE false THEN heavyMachine := stop
END
Loss of behaviour

Why is this problematic?

MACHINE $A$ ...
EVENT $emergencyStop \triangleq$
WHERE $true$ THEN $heavyMachine := stop$
END

refined by

MACHINE $R$ ...
EVENT $emergencyStop \triangleq$ REFINES $emergencyStop$
WHERE $false$ THEN $heavyMachine := stop$
END
Loss of behaviour

Why is this problematic?

MACHINE A ... 
EVENT emergencyStop ≡
WHERE true THEN heavyMachine := stop
END

refined by

MACHINE R ... 
EVENT emergencyStop ≡ REFINES emergencyStop
WHERE false THEN heavyMachine := stop
END

\[ E_{R;\text{evt}} = \emptyset \quad \Longrightarrow \quad R \text{ refines } A \]
Loss of behaviour

Every trace for $A$ has a refining trace for $R$. 
Loss of behaviour

Every trace for $A$ has a refining trace for $R$.

More precisely

For every trace in $A$ with triggered events $evt_{A,1}$, $evt_{A,2}$, \ldots, there is a trace in $R$ with triggered events $evt_{R,1}$, $evt_{R,2}$, \ldots and $evt_{R;i}$ refines $evt_{A;i}$. 
Loss of behaviour

Every trace for $A$ has a refining trace for $R$.

More precisely

For every trace in $A$ with triggered events $evt_{A,1}, evt_{A,2}, \ldots$, there is a trace in $R$ with triggered events $evt_{R,1}, evt_{R,2}, \ldots$ and $evt_{R,i}$ refines $evt_{A,i}$.

Definition: Lockfree Refinement

Let $R, A$ be two machines with the same state space $S$. $R$ is called a lockfree refinement of $A$ if

1. $I_R \subseteq I_A$
Loss of behaviour

Every trace for $A$ has a refining trace for $R$.

More precisely
For every trace in $A$ with triggered events $evt_{A,1}, evt_{A,2}, \ldots$, there is a trace in $R$ with triggered events $evt_{R,1}, evt_{R,2}, \ldots$ and $evt_{R;i}$ refines $evt_{A;i}$.

Definition: Lockfree Refinement
Let $R, A$ be two machines with the same state space $S$. $R$ is called a lockfree refinement of $A$ if

1. $I_R \subseteq I_A$
2. $I_R \neq \emptyset$
Loss of behaviour

Every trace for $A$ has a refining trace for $R$.

More precisely

For every trace in $A$ with triggered events $evt_{A,1}, evt_{A,2}, \ldots$, there is a trace in $R$ with triggered events $evt_{R,1}, evt_{R,2}, \ldots$ and $evt_{R;i}$ refines $evt_{A;i}$.

Definition: Lockfree Refinement

Let $R, A$ be two machines with the same state space $S$. $R$ is called a lockfree refinement of $A$ if

1. $I_R \subseteq I_A$
2. $I_R \neq \emptyset$
3. $E_{R;evt_R} \subseteq E_{A;evt_A}$ for each event
Loss of behaviour

Every trace for $A$ has a refining trace for $R$.

More precisely

For every trace in $A$ with triggered events $\text{evt}_{A,1}, \text{evt}_{A,2}, \ldots$, there is a trace in $R$ with triggered events $\text{evt}_{R,1}, \text{evt}_{R,2}, \ldots$ and $\text{evt}_{R;i}$ refines $\text{evt}_{A;i}$.

Definition: Lockfree Refinement

Let $R, A$ be two machines with the same state space $S$. $R$ is called a lockfree refinement of $A$ if

1. $I_R \subseteq I_A$
2. $I_R \neq \emptyset$
3. $E_{R;\text{evt}_R} \subseteq E_{A;\text{evt}_A}$ for each event
4. $\text{dom}(E_{A;\text{evt}_A}) \subseteq \text{dom}(E_{R;\text{evt}_R})$ for each event
Coupling

More general notion of refinement

What if abstract machine $A$ and refinement $R$ have different state spaces $S_A$ and $S_R$?

Couple abstract and refined state space.

$C \in S_R \leftrightarrow S_A$

Coupling invariant / Gluing invariant

Example

MACHINE AbstractFileSys
VARIABLES openFiles
INVARIANTS openFiles $\subseteq$ FILES

MACHINE RefinedFileSys
VARIABLES openModes
INVARIANTS openModes $\subseteq$ FILES $\times$ MODES

$C = \{ r \mapsto a | a = \text{dom}(r) \} = \{ f, m \cdot (f \mapsto m) \mapsto m \}$
Coupling

More general notion of refinement

What if abstract machine $A$ and refinement $R$ have different state spaces $S_A$ and $S_R$?

→ **Couple** abstract and refined state space.

$$C \in S_R \leftrightarrow S_A$$

**Coupling invariant / Gluing invariant**
Coupling

More general notion of refinement

What if abstract machine $A$ and refinement $R$ have different state spaces $S_A$ and $S_R$?

→ **Couple** abstract and refined state space.

$C \in S_R \leftrightarrow S_A$ **Coupling invariant / Gluing invariant**

**Example**

<table>
<thead>
<tr>
<th>MACHINE</th>
<th>AbstractFileSys</th>
<th>MACHINE</th>
<th>RefinedFileSys</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLES</td>
<td>openFiles</td>
<td>VARIABLES</td>
<td>openModes</td>
</tr>
<tr>
<td>INVARIANTS</td>
<td>openFiles $\subseteq$ FILES</td>
<td>INVARIANTS</td>
<td>openModes $\subseteq$ FILES $\times$ MODES</td>
</tr>
</tbody>
</table>
Coupling

More general notion of refinement

What if abstract machine $A$ and refinement $R$ have different state spaces $S_A$ and $S_R$?

→ **Couple** abstract and refined state space.

\[ C \in S_R \leftrightarrow S_A \]

**Coupling invariant / Gluing invariant**

### Example

**MACHINE** *AbstractFileSys*

**VARIABLES** *openFiles*

**INVARIANTS**

\[ openFiles \subseteq FILES \]

**MACHINE** *RefinedFileSys*

**VARIABLES** *openModes*

**INVARIANT**

\[ openModes \subseteq FILES \times MODES \]

\[ C = \{ r \mapsto a \mid a = \text{dom}(r) \} = \{ f, m \cdot (f \mapsto m) \mapsto m \} \]
Refinement – Coupling

- Sensible to assume $C$ a total relation:

$$C \in S_R \iff S_A$$

- Often, coupling is a total function:

$$C \in S_R \rightarrow S_A$$

Define one abstraction for any detailed state.
BUT sometimes, several possible abstractions per concrete state sensible.
Refinement – Coupled Traces

Refinement: $R$ refines $A$
For every concrete trace $(\chi_0, \chi_1, ...)\) of $R$ with events $(\text{evt}_R^1, \text{evt}_R^2, ...)\) there exists an abstract trace $(\sigma_0, \sigma_1, ...)\) with events $(\text{evt}_A^1, \text{evt}_A^2, ...)\) such that $\chi_i \mapsto \sigma_i \in C$ for all $i \in \mathbb{N}$
$\text{evt}_R^i$ refines event $\text{evt}_A^i$. 
Refinement – Coupled Traces

For every concrete trace \((\chi_0, \chi_1, \ldots)\) of \(R\) with events \((\text{evt}_R^1, \text{evt}_R^2, \ldots)\) there exists an abstract trace \((\sigma_0, \sigma_1, \ldots)\) with events \((\text{evt}_A^1, \text{evt}_A^2, \ldots)\) such that

\[
\chi_i \mapsto \sigma_i \in C \quad \text{for all } i \in \mathbb{N}
\]

\(\text{evt}_R^i\) refines event \(\text{evt}_A^i\).
Refinement: $R$ refines $A$

For every concrete trace $(\chi_0, \chi_1, \ldots)$ of $R$ with events $(\text{evt}_1^R, \text{evt}_2^R, \ldots)$ there exists an abstract trace $(\sigma_0, \sigma_1, \ldots)$ with events $(\text{evt}_1^A, \text{evt}_2^A, \ldots)$ such that

$$
\chi_i \mapsto \sigma_i \in C \quad \text{for all } i \in \mathbb{N}
$$

$$
\text{evt}_i^R \text{ refines event } \text{evt}_i^A.
$$
Refinement: $R$ refines $A$

For every concrete trace $(\chi_0, \chi_1, \ldots)$ of $R$ with events $(\mathrm{evt}_1^R, \mathrm{evt}_2^R, \ldots)$ there exists an abstract trace $(\sigma_0, \sigma_1, \ldots)$ with events $(\mathrm{evt}_1^A, \mathrm{evt}_2^A, \ldots)$ such that

$$\chi_i \mapsto \sigma_i \in C \text{ for all } i \in \mathbb{N}$$
Refinement: $R$ refines $A$

For every concrete trace $(\chi_0, \chi_1, \ldots)$ of $R$ with events $(\text{evt}_1^R, \text{evt}_2^R, \ldots)$ there exists an abstract trace $(\sigma_0, \sigma_1, \ldots)$ with events $(\text{evt}_1^A, \text{evt}_2^A, \ldots)$ such that

1. $\chi_i \mapsto \sigma_i \in C$ for all $i \in \mathbb{N}$
2. $\text{evt}_i^R$ refines event $\text{evt}_i^A$. 
Refinement – Definition

**Definition: Refinement**

Let $R, A$ be two machines with state spaces $S_R, S_A$. Let $C \in S_R \leftrightarrow R_A$ be the coupling invariant. $R$ is called a refinement of $A$ modulo $C$ if

1. \( I_R \subseteq C - I_A \)
2. \( E_R \cdot \text{evt}_R \subseteq C \cdot \text{evt}_A \cdot C^{-1} \) for each event.
Refinement – Definition

Definition: Refinement

Let $R, A$ be two machines with state spaces $S_R, S_A$. Let $C \in S_R \leftrightarrow R_A$ be the coupling invariant. $R$ is called a refinement of $A$ modulo $C$ if

1. $I_R \subseteq C^{-1}[I_A]$ and

   $(\forall x, y \cdot x \leftrightarrow y \in R^{-1} \iff y \leftrightarrow x \in R$, inverse relation)
Definition: Refinement

Let $R, A$ be two machines with state spaces $S_R, S_A$. Let $C \in S_R \leftrightarrow R_A$ be the coupling invariant. $R$ is called a refinement of $A$ modulo $C$ if

1. $I_R \subseteq C^{-1}[I_A]$ and
2. $E_{R;evt_R} \subseteq C \cdot E_{A;evt_A} \cdot C^{-1}$ for each event.

($\forall x, y \cdot x \leftrightarrow y \in R^{-1} \iff y \leftrightarrow x \in R$, inverse relation)
Refinement – Path subsumption

\[ \sigma_n \xrightarrow{\text{evt}_A} \sigma_{n+1} \]

\[ C \downarrow \]

\[ \chi_n \xrightarrow{\text{evt}_R} \chi_{n+1} \]

\[ C \uparrow \]
Refinement – Path subsumption

\[ \sigma_n \xrightarrow{\text{evt}_A} \sigma_{n+1} \]

\[ \chi_n \xrightarrow{\text{evt}_R} \chi_{n+1} \]

\[ E_R; \text{evt}_R \subseteq C ; E_A; \text{evt}_A ; C^{-1} \]
Refinement – Path subsumption

\[
\sigma_n \xrightarrow{\text{evt}_A} \sigma_{n+1}
\]

\[
\chi_n \xleftarrow{\text{evt}_R} \chi_{n+1}
\]

\[
E_{R;\text{evt}_R} \subseteq C ; E_{A;\text{evt}_A} ; C^{-1}
\]
Refinement – Path subsumption

\[ E_{R;\text{evt}_R} \subseteq C ; E_{A;\text{evt}_A} ; C^{-1} \]
Refinement – Path subsumption

\[ \sigma_n \xrightarrow{\text{evt}_A} \sigma_{n+1} \]

\[ \chi_n \subset \subset \chi_{n+1} \]

\[ E_R; \text{evt}_R \subseteq C ; E_A; \text{evt}_A ; C^{-1} \]
The coupling invariant is specified as part of the invariant of the refining machine.
The coupling invariant is specified as part of the invariant of the refining machine.

The invariant of a refinement is allowed to refer to variables of its abstraction.
Specifying Coupling

The coupling invariant is specified as part of the invariant of the refining machine.

The invariant of a refinement is allowed to refer to variables of its abstraction.

Example (from slide 72)

**MACHINE AbstractFileSys**

**VARIABLES** openFiles

**INVARIANTS**

\[ \text{openFiles} \subseteq \text{FILES} \]

**MACHINE RefinedFileSys**

**VARIABLES** openModes

**INVARIANTS**

\[ \text{openModes} \subseteq \text{FILES} \times \text{MODES} \]
Specifying Coupling

The coupling invariant is specified as part of the invariant of the refining machine.

The invariant of a refinement is allowed to refer to variables of its abstraction.

Example (from slide 72)

MACHINE AbstractFileSys
VARIABLES openFiles
INVARIANTS
  openFiles ⊆ FILES

MACHINE RefinedFileSys
VARIABLES openModes
INVARIANTS
  openModes ⊆ FILES × MODES
  openFiles = dom(openModes)
Proof Obligation GRD

Proof that event guard in refinement is stronger than in abstract machine.
⇒ Abstraction is enabled when refinement is.

Abstract invariants
Concrete invariants
Concrete event guard
⇒
Abstract event guard

∀ vars A, vars R · inv A (vars A) ∧ inv R (vars A, vars R) ∧ grd R (vars R) ⇒ grd A (vars A) (Version w/o parameters, see literature for full version)
Proof Obligation GRD

Proof that event guard in refinement is **stronger** than in abstract machine.

\[ \Rightarrow \text{Abstraction is enabled when refinement is.} \]

Abstract invariants

Concrete invariants

Concrete event guard

\[ \Rightarrow \]

Abstract event guard

\[ \forall \text{vars}_A, \text{vars}_R \cdot \]

\[ \text{inv}_A(\text{vars}_A) \land \text{inv}_R(\text{vars}_A, \text{vars}_R) \land \text{grd}_R(\text{vars}_R) \]

\[ \Rightarrow \text{grd}_A(\text{vars}_A) \]

(Version w/o parameters, see literature for full version)
Show that refined action *simulates* abstract actions.

Abstract invariants
Concrete invariants
Concrete event guard
Concrete before-after-predicate

⇒
Abstract before-after-predicate
Proof Obligation SIM

Show that refined action *simulates* abstract actions

Abstract invariants
Concrete invariants
Concrete event guard
Concrete before-after-predicate

\[ \Rightarrow \]

Abstract before-after-predicate

**Rem**  \( E_{R;evt_R} \subseteq C ; E_{A;evt_A} ; C^{-1} \)
Show that refined action simulates abstract actions

Abstract invariants
Concrete invariants
Concrete event guard
Concrete before-after-predicate

⇒
Abstract before-after-predicate

Rem $E_{R;evt_R} \subseteq C ; E_{A;evt_A} \cdot C^{-1}$

Obs The coupling invariant is only used for the before-state not for the after-state.
Proof Obligation **SIM**

Show that refined action *simulates* abstract actions

Abstract invariants
Concrete invariants
Concrete event guard
Concrete before-after-predicate

$$\implies$$
Abstract before-after-predicate

**Rem** \( E_{R';evt_R} \subseteq C ; E_{A';evt_A} ; C^{-1} \)

**Obs** The coupling invariant is only used for the before-state not for the after-state.

? Why?
Proof Obligation SIM

Show that refined action *simulates* abstract actions

Abstract invariants
Concrete invariants
Concrete event guard
Concrete before-after-predicate

⇒

Abstract before-after-predicate

Rem \( E_{R;\text{evt}_R} \subseteq C ; E_{A;\text{evt}_A} ; C^{-1} \)

Obs The coupling invariant is only used for the before-state not for the after-state.

Why?

Already proven condition \( \text{INV} \) implies invariant for after-state.
Event-B has more ...

Things not covered in these slides:

- Witnesses for parameters dropped in refinements
- Termination issues (variants)
- Extended/Not extended events
- Event merging
- Sequential refinement
- ...

Klebanov, Ulbrich – Applications of Formal Verification
Event-B has more ...

Things not covered in these slides:

- Witnesses for parameters dropped in refinements
- Termination issues (variants)
- Extended/Not extended events
- Event merging
- Sequential refinement
- ...
Event-B has more ...

Things not covered in these slides:

- Witnesses for parameters dropped in refinements
- Termination issues (variants)
- Extended/Not extended events
- Event merging
- Sequential refinement
- ...

Klebanov, Ulbrich – Applications of Formal Verification
Event-B has more ...

Things not covered in these slides:

- Witnesses for parameters dropped in refinements
- Termination issues (variants)
- Extended/Not extended events
- Event merging
- Sequential refinement
- ...
Event-B has more ...

Things not covered in these slides:

- Witnesses for parameters dropped in refinements
- Termination issues (variants)
- Extended/Not extended events
- Event merging
- Sequential refinement
- ...
Event-B has more ...

Things not covered in these slides:

- Witnesses for parameters dropped in refinements
- Termination issues (variants)
- Extended/Not extended events
- Event merging
- Sequential refinement
- ...
Byzantine Agreement –
A case study verified with Event-B

Based on:
Byzantine Generals

"When shall we attack?"
Byzantine Generals

“When shall we attack?”
Byzantine Generals

“When shall we attack?”

messages
Byzantine Generals

"When shall we attack?"

agree on a time even in the presence of traitors
Application in Avionics

Which components are operative?

C1, C2, C3, C4 agree on the set of operative components even in the presence of faulty components.
Application in Avionics

“Which components are operative?”

C1
C2
C3
C4
Application in Avionics

“Which components are operative?”

agree on the set of **operative components** even in the presence of **faulty components**
Explanation by Example

C1
C2
C3
C4

CONSENSUS!
Explanation by Example

C2 → C1
1

C2 → C3
2

C2 → C4
3

CONSENSUS!
Explanation by Example

C2 → C1
1 → 1

C2 → C3
2 → 2

C2 → C4
3 → 3

C1 → C4
3 → 3

C3 → C4
1,3,2 → 2

C1,2,3 → C4
2,1,3 → 3
Explanation by Example
Explanation by Example

C1

C2

C3

C4

1

1

1

2

3

CONSENSUS!

Klebanov, Ulbrich – Applications of Formal Verification
Explanation by Example

C2

C1

C3

C4

CONSENSUS!
Explanation by Example

C1

C2

C3

C4

CONSENSUS!
Explanation by Example

C1 → C4
C2 → C3 → C4

C1

C2

C3

C4

1,3,2
3
2

2,1,3

3,1,2
Explanation by Example

C1

C2

C3

C4

1,3,2

2,1,3

3,1,2

CONSENSUS!
Example Run 2
Example Run 2

Round 0

C1

C2

C3

C4

Klebanov, Ulbrich – Applications of Formal Verification

SS 2015

85/96
Example Run 2

Round 1

C1

C2

C3

C4

Klebanov, Ulbrich – Applications of Formal Verification
Example Run 2

Round 0

Round 1

Round 2

Klebanov, Ulbrich – Applications of Formal Verification
Byzantine Agreement Algorithm

Verification Goals:

Validity If the transmitter $tt$ is non-faulty, then all non-faulty receivers agree on the value sent by $tt$.

Agreement Any two non-faulty receivers agree on the value assigned to $tt$. 
Byzantine Agreement Algorithm

Round 0: Transmitter sends signed message to all receivers.
Byzantine Agreement Algorithm

Round 0: Transmitter sends signed message to all receivers.

Round \( n \): Component receive messages, verify signatures, sign messages and pass them on.
Byzantine Agreement Algorithm

Round 0: Transmitter sends signed message to all receivers.

Round $n$: Component receive messages, verify signatures, sign messages and pass them on.

GOAL: Prove that this algorithm has the “validity” and “agreement” properties.
Verification

Quote

*We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm.*

Taken from: *The Byzantine Generals Problem*

Leslie Lamport, Robert Shostak, and Marshall Pease
ACM Transactions on Programming Languages and Systems
CONTEXT Context
SETS

CONSTANTS

AXIOMS

END
Context for Byzantine Agreement

**CONTEXT** Context

SETS

MODULE

VALUE

CONSTANTS

AXIOMS

END
CONTEXT Context
SETS
   MODULE
   VALUE
CONSTANTS
   faulty, transmitter, V₀
AXIOMS

END
CONTEXT Context

SETS
  MODULE
  VALUE

CONSTANTS
  faulty, transmitter, V_0

AXIOMS
  faulty ⊆ MODULE
  transmitter ∈ MODULE
  V_0 ∈ VALUE
  finite(faulty)

END
First machine

MACHINE Messages
SEES Context

VARIABLES

INVARIANTS

...
First machine

MACHINE Messages
SEES Context

VARIABLES messages, round, collected

INVARIANTS

...
First machine

MACHINE Messages
SEES Context

VARIABLES \textit{messages}, \textit{round}, \textit{collected}

ININVARIANTS
\begin{align*}
\text{ty} \_ \text{mess} : \textit{messages} \subseteq \textit{MODULE} \times \textit{MODULE} \times \textit{VALUE}
\end{align*}

\ldots

\textit{messages} messages being sent in the \textit{current} round
MACHINE Messages
SEES Context

VARIABLES messages, round, collected

INVARIANTS
  ty_mess : messages ⊆ Module × Module × Value
  ty_round : round ∈ N

messages messages being sent in the current round
round the number of the current round
First machine

MACHINE Messages
SEES Context

VARIABLES messages, round, collected

INVPARIANTS

\[ \text{ty\_mess} : \text{messages} \subseteq \text{MODULE} \times \text{MODULE} \times \text{VALUE} \]
\[ \text{ty\_round} : \text{round} \in \mathbb{N} \]
\[ \text{ty\_collected} : \text{collected} \in \text{MODULE} \rightarrow \mathbb{P}(\text{VALUE}) \]

\[ \ldots \]

\textit{messages} messages being sent in the current round
\textit{round} the number of the current round
\textit{collected} values observed in previous rounds
First machine (2)

*messages*  messages being sent in the *current* round

*round*  the number of the current round

*collected*  values observed in previous rounds
First machine (2)

messages messages being sent in the current round
round the number of the current round
collected values observed in previous rounds

MACHINE Messages SEES Context
VARIABLES messages, round, collected
INvariants...
EVENTS
   Initialisation ≡ ...
   EVENT ROUND ≡

END
First machine (2)

- **messages** messages being sent in the *current* round
- **round** the number of the current round
- **collected** values observed in previous rounds

```
MACHINE Messages SEES Context
VARIABLES messages, round, collected
INVARIANTS...
EVENTS
  Initialisation \equiv ... 
  EVENT ROUND \equiv 
    act1 : round := round + 1
END
```
First machine (2)

messages messages being sent in the current round
round the number of the current round
collected values observed in previous rounds

MACHINE Messages SEES Context
VARIABLES messages, round, collected
INVESTIGANTS...
EVENTS

Initialisation  \equiv \ldots

EVENT ROUND \equiv
  act1 : round := round + 1
  act2 : messages \in IP(Module \times Module \times Value)
First machine (2)

messages messages being sent in the current round
round the number of the current round
collected values observed in previous rounds

MACHINE Messages SEES Context
VARIABLES messages, round, collected
INVARIANTS...
EVENTS

Initialisation \( \cong \ldots \)

EVENT ROUND \( \cong \)
act1 : round \( \cong \) round + 1
act2 : messages \( \in \) \( \mathbb{P}(\text{MODULE} \setminus \{\text{transmitter}\} \times \text{MODULE} \times \text{VALUE}) \)

END
First machine (2)

**messages** messages being sent in the *current* round

**round** the number of the current round

**collected** values observed in previous rounds

---

**MACHINE** *Messages* **SEES** *Context*

**VARIABLES** *messages*, *round*, *collected*

**INVARIANTS**...

**EVENTS**

*Initialisation* ≡ ...

**EVENT** **ROUND** ≡

- **act1** : *round* := *round* + 1
- **act2** : *messages* ∈ \(P(MODULE \setminus \{transmitter\} \times MODULE \times VALUE)\)
- **act3** : *collected* := \(\lambda m \cdot collected(m) \cup \)

**END**
First machine (2)

messages messages being sent in the current round

round the number of the current round

collected values observed in previous rounds

MACHINE Messages SEES Context

VARIABLES messages, round, collected

INVARIANTS...

EVENTS

Initialisation ≡ ...

EVENT ROUND ≡

act1 : round := round + 1

act2 : messages ∈ IP(Module \ \{transmitter\} × Module × Value)

act3 : collected := λm · collected(m) ∪ \{v | (s, m, v) ∈ messages\}

END
First refinement: signed messages

All messages are signed in a trustworthy manner:

No forgery possible $\implies$ Consider only *relayed* messages.
First refinement: signed messages

All messages are signed in a trustworthy manner:

No forgery possible \(\implies\) Consider only \textbf{relayed} messages.

round \(k\):

\[
\begin{align*}
\text{s} & \quad \rightarrow \quad \text{v} \\
& \downarrow \\
\text{r}
\end{align*}
\]
First refinement: signed messages

All messages are signed in a trustworthy manner:

No forgery possible \(\implies\) Consider only \textit{relayed} messages.

round \(k\):

\[
s \xrightarrow{v} r
\]

round \(k + 1\):

\[
r \xrightarrow{v} n
\]
Signed messages (2)

round \( k \): \( s \rightarrow^v r \)

round \( k + 1 \): \( r \rightarrow^v n \)

MACHINE SignedMessages \ REFINES Messages
VARIABLES messages, round, collected
INVARIANTS

EVENTS

END
Signed messages (2)

round $k$: $s \xrightarrow{v} r$
round $k + 1$: $r \xrightarrow{v} n$

MACHINE SignedMessages REFINES Messages
VARIABLES messages, round, collected
INVARIANTS

EVENTS
EVENT ROUND REFINES ROUND $\equiv$
    act1, act3 as above

END
Signed messages (2)

\[ \text{round } k: \quad s \rightarrow^v r \]
\[ \text{round } k + 1: \quad r \rightarrow^v n \]

\[
\begin{align*}
\text{MACHINE } & \text{SignedMessages } \text{ REFINES Messages} \\
\text{VARIABLES } & \text{messages, round, collected} \\
\text{INVARIANTS } & \\
\text{EVENTS } & \\
\text{EVENT } & \text{ROUND REFINES ROUND } \cong \\
& \text{act1, act3 as above} \\
\text{was: messages } & : \in \mathcal{P}((\text{MODULE } \setminus \{\text{transmitter}\} \times \text{MODULE } \times \text{VALUE})) \\
\text{END }
\end{align*}
\]
Signed messages (2)

round $k$: $s$ \xrightarrow{v} r
round $k + 1$: $r$ \xrightarrow{v} n

MACHINE SignedMessages REFINES Messages

VARIABLES messages, round, collected

INVARIANTS

EVENTS

EVENT $ROUND \text{ REFINES } ROUND \triangleq$
act1, act3 as above
act2: $messages \in \mathcal{P}\left(\{(r, n, v) \mid (s, r, v) \in messages\}\right)$
was : $messages \in \mathcal{P}(\text{MODULE} \setminus \{\text{transmitter}\} \times \text{MODULE} \times \text{VALUE})$

END
Signed messages (2)

round \( k \): \( s \xrightarrow{v} r \)

round \( k + 1 \): \( r \xrightarrow{v} n \)

**MACHINE** `SignedMessages` **REFINES** `Messages`

**VARIABLES** `messages`, `round`, `collected`

**INVARIANTS**

val1: \( \forall s, r, v \cdot (s, r, v) \in messages \Rightarrow v \in collected(\text{transmitter}) \)

**EVENTS**

**EVENT** `ROUND` **REFINES** `ROUND` \( \equiv \)

act1, act3 as above

act2: `messages` :\( \in \mathbb{P} \left( \left\{ (r, n, v) \mid (s, r, v) \in messages \right\} \right) \)

was : `messages` :\( \in \mathbb{P}(\text{MODULE \{transmitter\} \times MODULE \times VALUE}) \)

**END**
Signed messages (2)

\[
\begin{align*}
\text{round } k: & \quad s \xrightarrow{v} r \\
\text{round } k + 1: & \quad r \xrightarrow{v} n
\end{align*}
\]

\begin{environment}
\text{MACHINE} \text{SignedMessages} \text{ REFINES} \text{ Messages}
\text{VARIABLES} \text{messages, round, collected}
\text{INVARIANTS}
\quad \text{val1:} \forall s, r, v \cdot (s, r, v) \in \text{messages} \Rightarrow v \in \text{collected(\text{transmitter})}
\quad \text{val2:} \forall n \cdot \text{collected}(n) \subseteq \text{collected(\text{transmitter})}
\text{EVENTS}
\text{EVENT ROUND REFINES ROUND } \cong \\
\quad \text{act1, act3 } \text{as above}
\quad \text{act2:} \text{messages} :\in \mathbb{P}(\{(r, n, v) \mid (s, r, v) \in \text{messages}\})
\quad \text{was :} \text{messages} :\in \mathbb{P}((\text{MODULE} \setminus \{\text{transmitter}\} \times \text{MODULE} \times \text{VALUE})
\end{environment}
Refinement Tower

Context

Messages

MessagesSigned

History

Guarantees

HybridGuarantees

GuaranteesTech

HybridGuaranteesTech

HybridContext

ValueTables

Roundless

ModuleList

VotingContext

ZA

ValueTablesTech

SM

Klebanov, Ulbrich – Applications of Formal Verification

SS 2015

94/96
Refinement Tower

covered so far

Messages
MessagesSigned
History
Guarantees
GuaranteesTech
HybridGuarantees
Roundless
HybridGuaranteesTech
ValueTables
ValueTablesTech
ZA
SM

HybridContext

VotingContext

ModuleList

Context
Refinement Tower

Changes message representation:
\[ \text{msgs} \subseteq \text{MODULE} \times \text{MODULE} \times \mathcal{P}(\text{MODULE}) \times \text{VALUE} \]
non-faulty modules behave well:

\[ r \notin \text{faulty} \land (s, r, h, v) \in \text{msgs} \implies \]
\[ \forall n \cdot (n \notin h \implies (r, n, h \cup \{r\}, v) \in \text{msgs}^{'}) \]
Refinement Tower

hybrid fault model:

\[ \text{faulty} = \text{arbFault} \cup \text{symFaulty} \]

\[ \text{arbFaulty} \cap \text{symFaulty} = \emptyset \]
new event structure:

```plaintext
PROCESS_EVENT refines SKIP
```

modifies internal data structures (invisible to abstract machine) and

```plaintext
ROUND_SWITCH refines ROUND
```

reproduces the effect of a round change from the internal data.

**An implementation would refine PROCESS_EVENT.**
Agreement!

In machine Guarantees:

\[ \text{round} \geq \text{card}(\text{faulty}) + 1 \implies \]

\[ (\forall n, m \cdot n \notin \text{faulty} \land m \notin \text{faulty} \implies \]

\[ \text{collected}(n) = \text{collected}(m) \]
Agreement!

In machine Guarantees:

\[
\text{round} \geq \text{card}(\text{faulty}) + 1 \implies \\
(\forall n, m \cdot n \notin \text{faulty} \land m \notin \text{faulty} \implies \\
\text{collected}(n) = \text{collected}(m))
\]

In machine HybridGuarantees:

\[
\text{round} \geq \text{card}(\text{arbFaulty}) + 1 \implies \\
(\forall n, m \cdot n \notin \text{faulty} \land m \notin \text{faulty} \implies \\
\text{collected}(n) = \text{collected}(m))
\]
Verification Effort

<table>
<thead>
<tr>
<th>Numbers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>4 contexts, 12 machines, 106 invariants</td>
</tr>
<tr>
<td>Labour</td>
<td>approx. 4 person months</td>
</tr>
<tr>
<td>Proofs</td>
<td>322 proof obligations</td>
</tr>
<tr>
<td>Automation</td>
<td>74/322, 23%</td>
</tr>
</tbody>
</table>