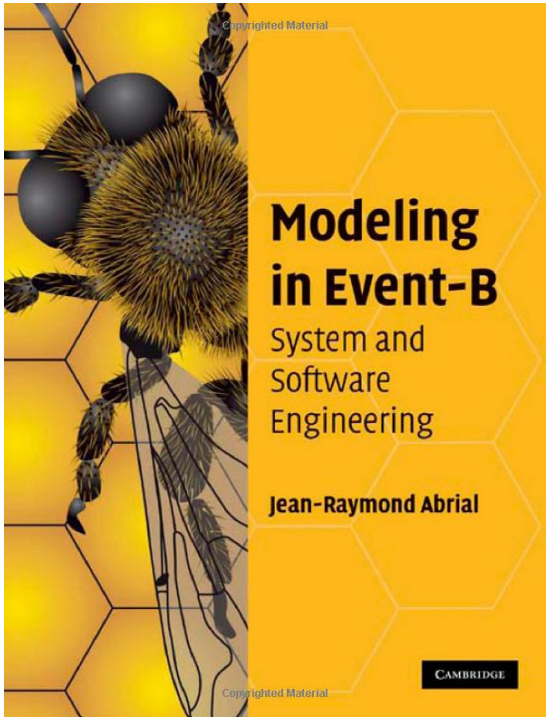
Applications of Formal Verification

Formal Software Design: Modelling in Event-B

Bernhard Beckert · Mattias Ulbrich | SS 2017

KIT – Institut für Theoretische Informatik

Literatur



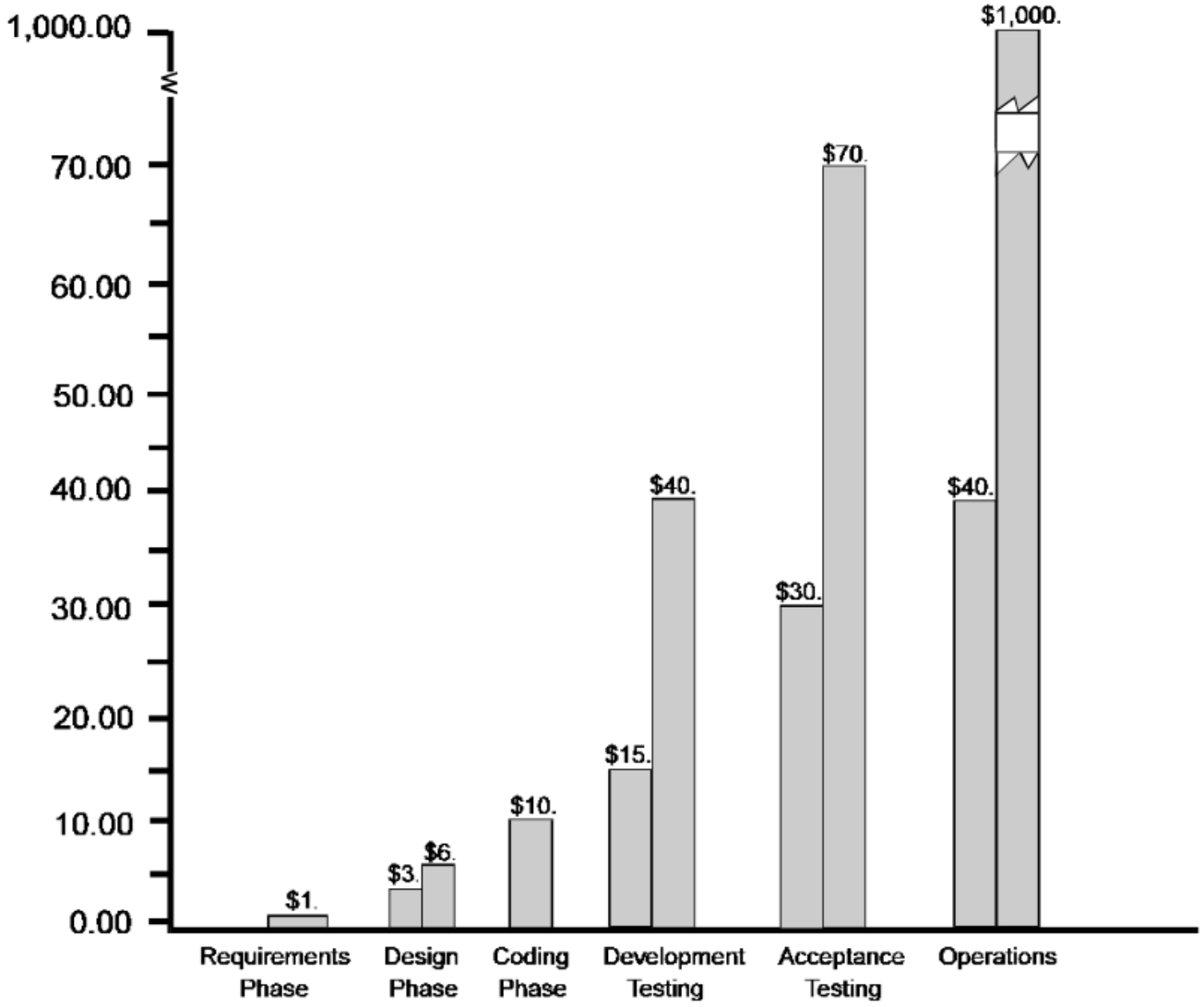
Jean-Raymond Abrial:
Modelling in Event-B
System and Software Enginieering
Cambridge University Press, 2010

Jean-Raymond Abrial:
The B-Book: Assigning programs to meanings
Cambridge University Press, 1996

Abstraction and Refinement – Introduction

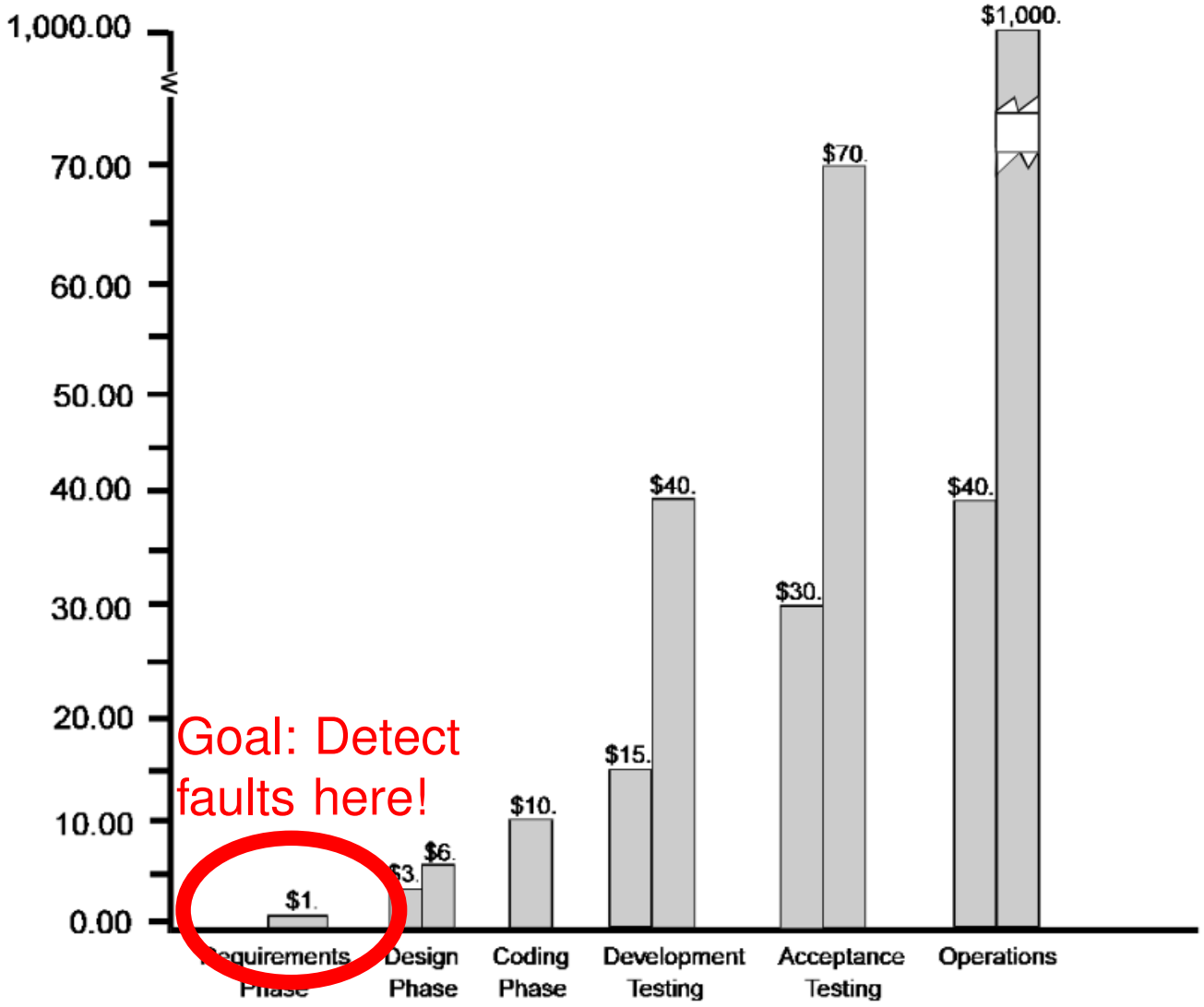
Late fault recovery is expensive

["Extra Time Saves Money", W. Knuffel, Computer Language, 1990]



Late fault recovery is expensive

["Extra Time Saves Money", W. Knuffel, Computer Language, 1990]



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

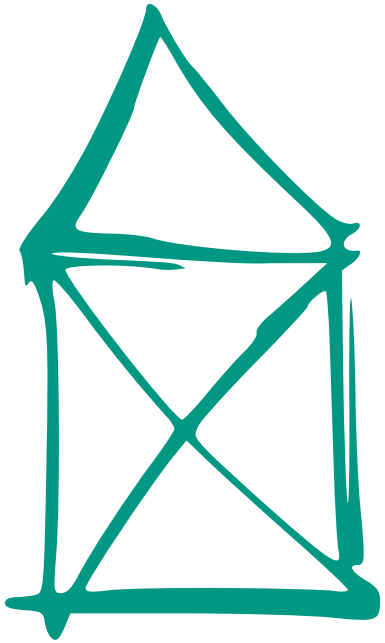
- Systems are **inherently complex**
- Unconsidered situations, **corner cases**
- **Ambiguous** natural language requirements
- Component **interplay**
- ...

Abstraction

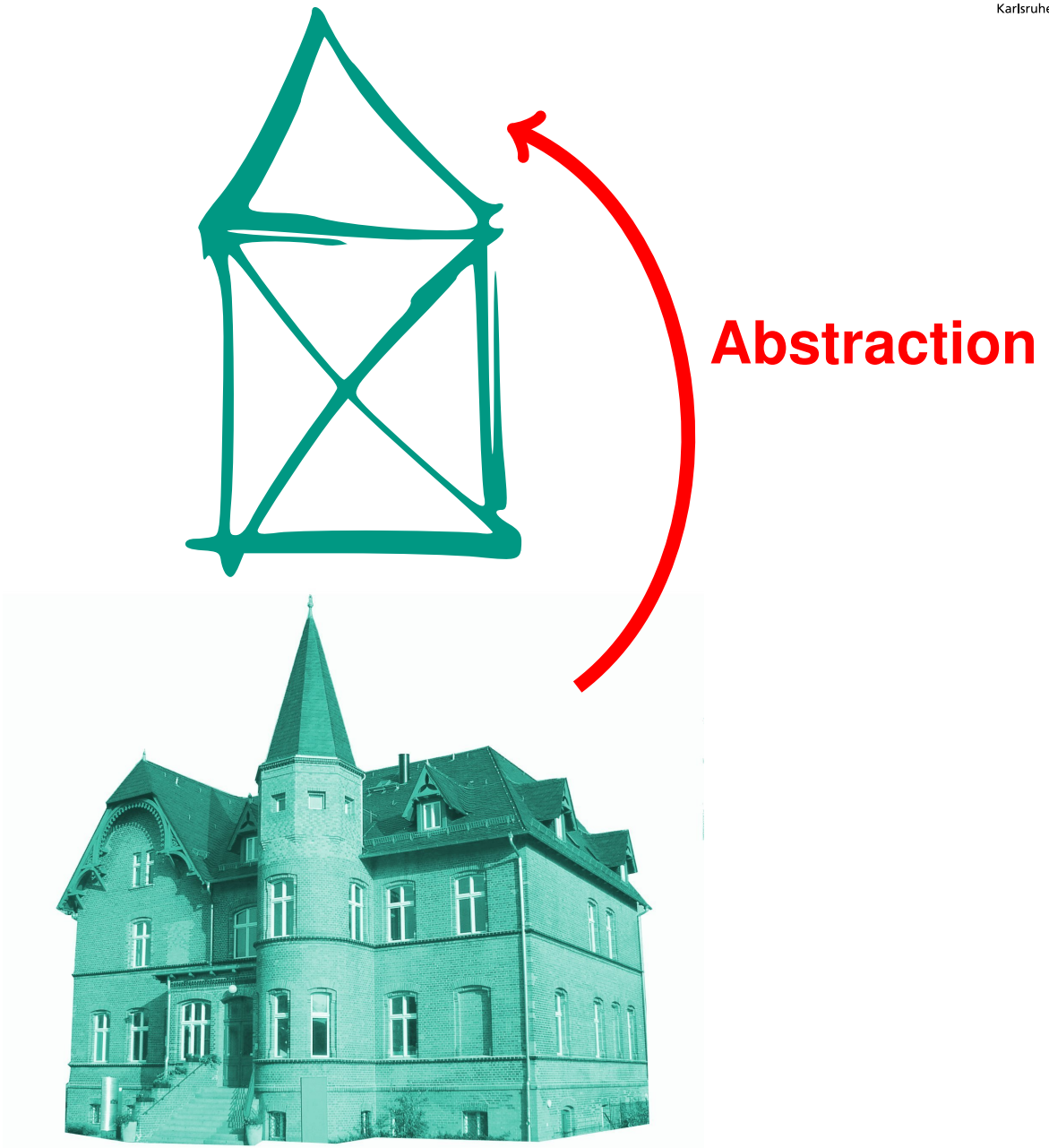
The only tool to **master complexity** is **abstraction**.

CLIFF JONES

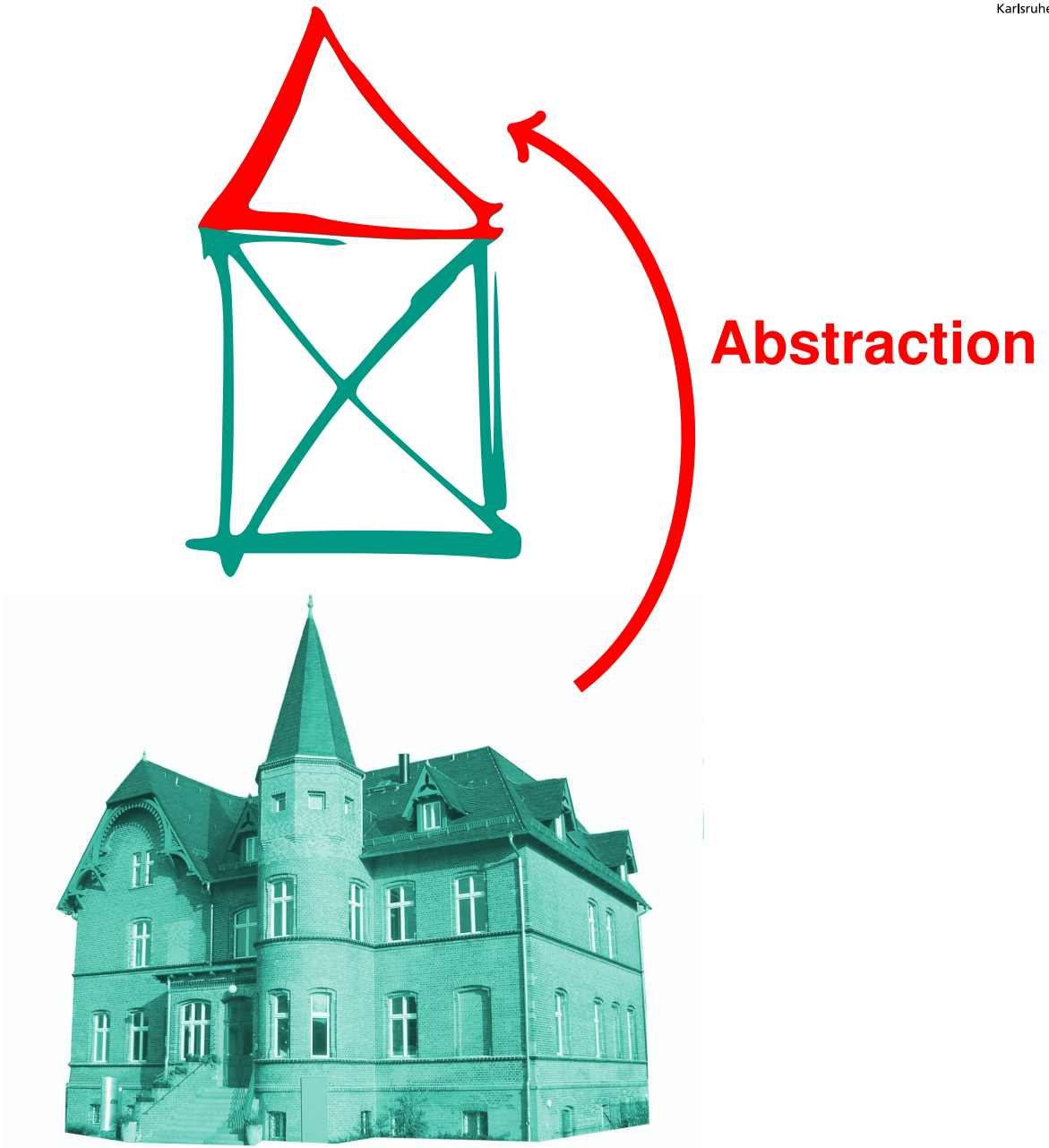
Abstraction and Refinement



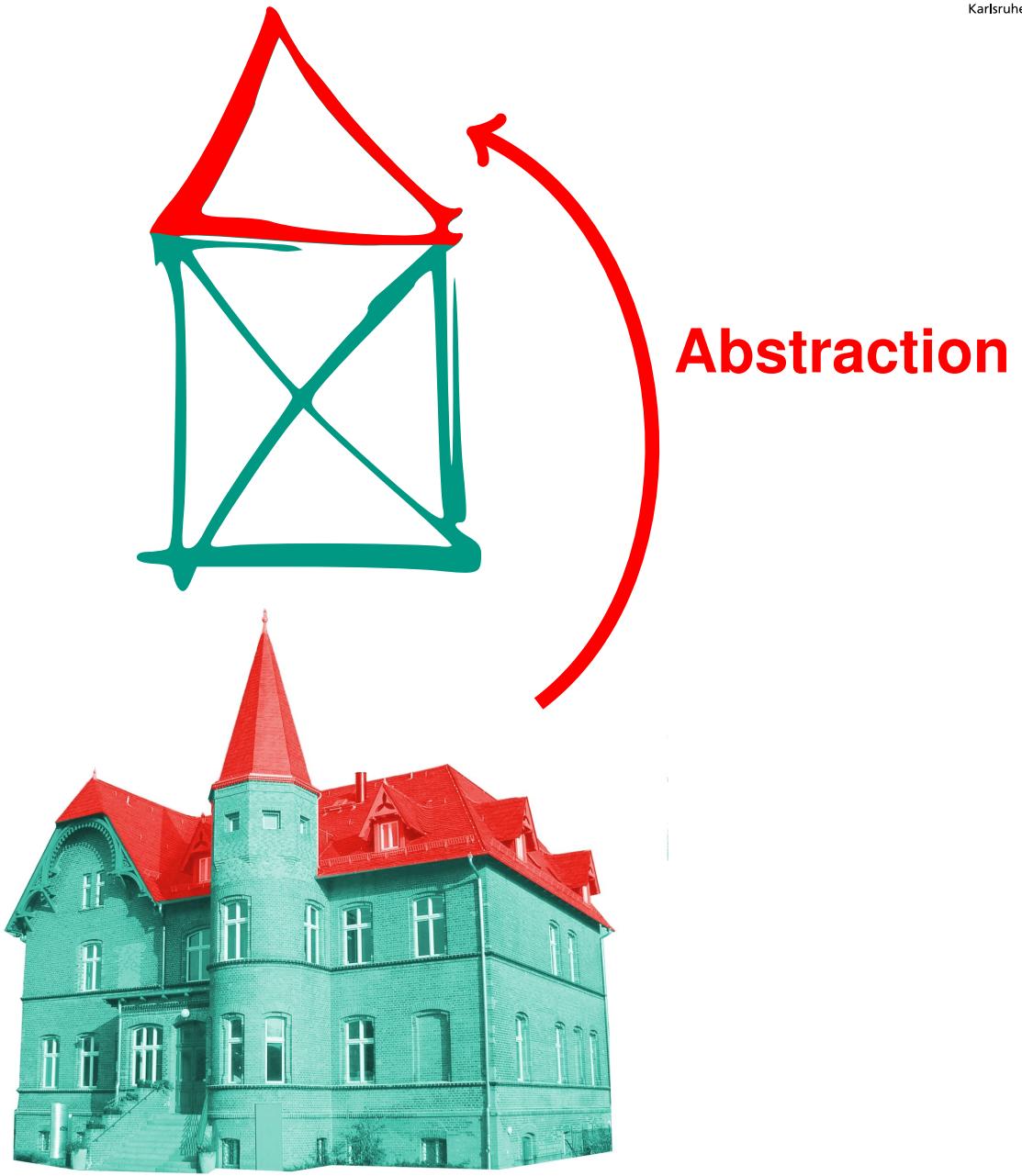
Abstraction and Refinement



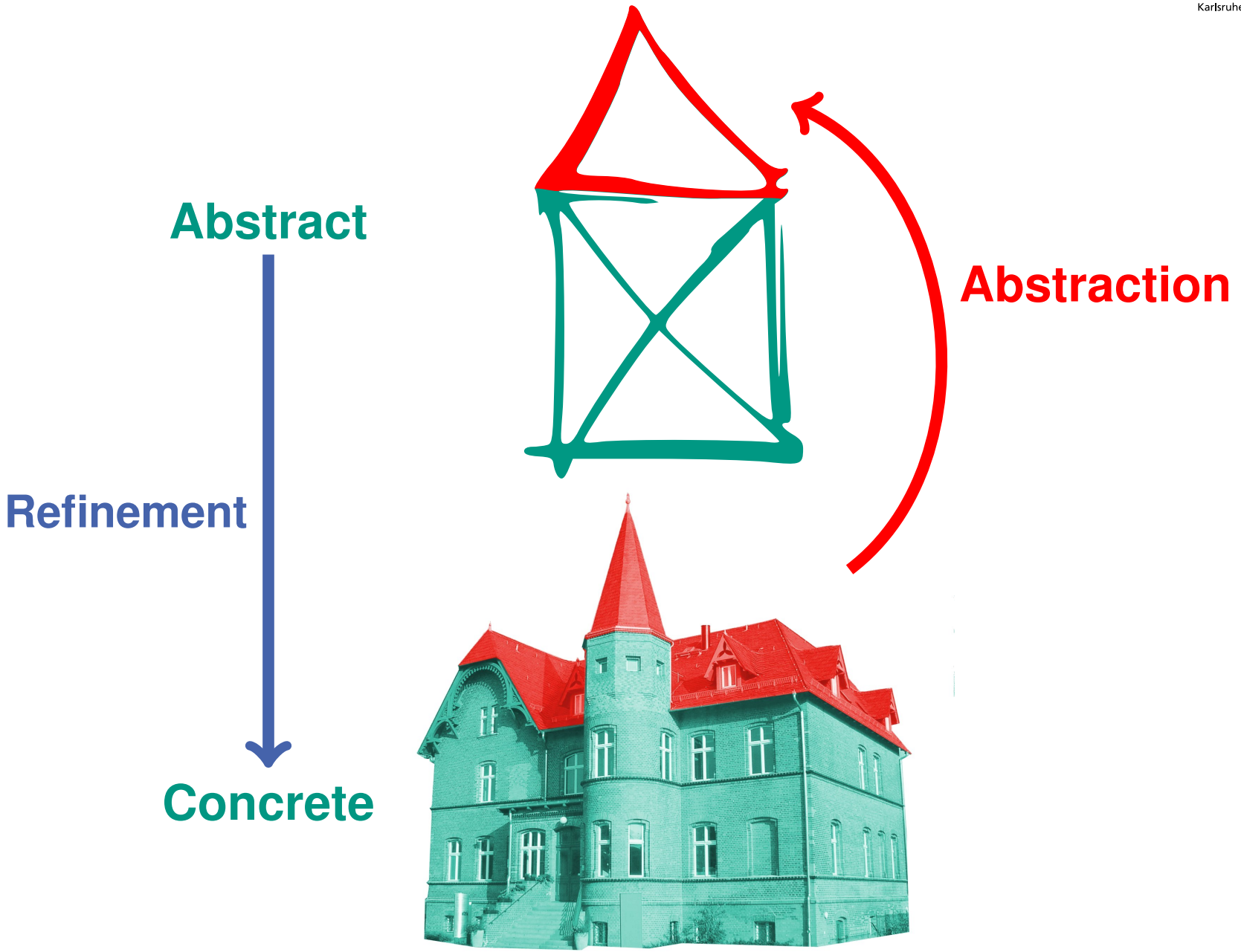
Abstraction and Refinement



Abstraction and Refinement



Abstraction and Refinement



Abstraction

Abstraction

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

and the inverse

Refinement

- enrich abstract model with details
- introduce a new particular aspect
- iterative process: add complexity in a stepwise fashion

Abstraction in Engineering

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

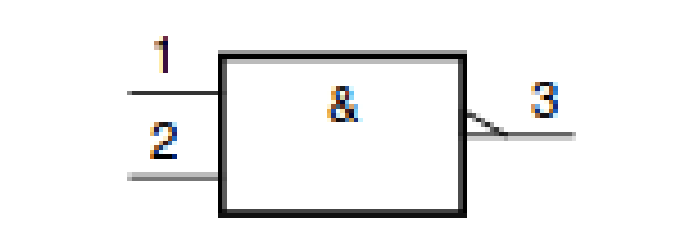
Aspect **Behaviour**

Aspect **Geometry**

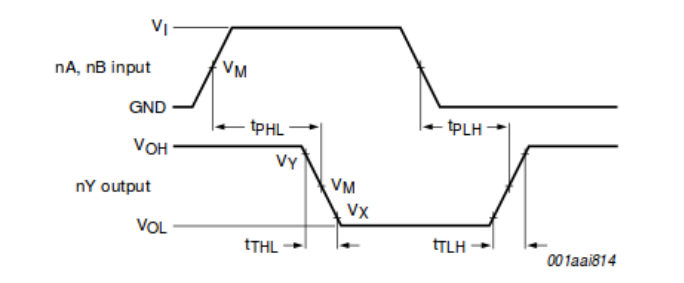
Datasheet – Abstraction

Extracts from datasheet for an IC with four NAND gates

Aspect **Behaviour**



refined to

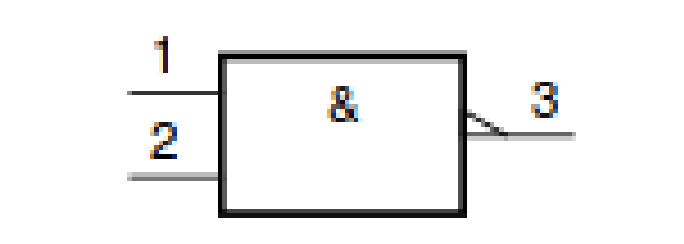


Aspect **Geometry**

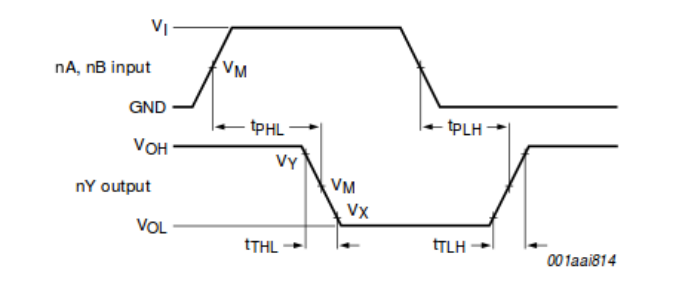
Datasheet – Abstraction

Extracts from datasheet for an IC with four NAND gates

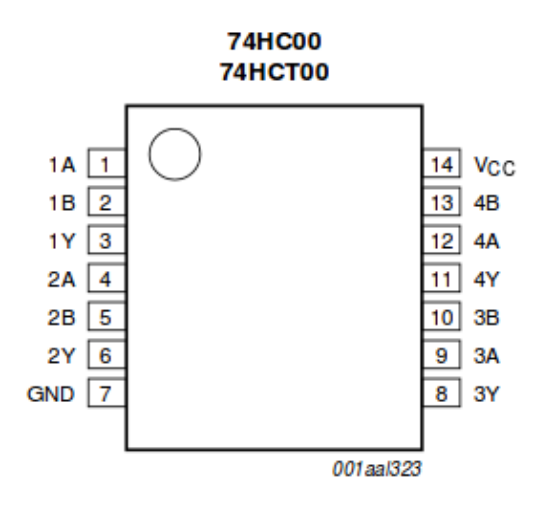
Aspect **Behaviour**



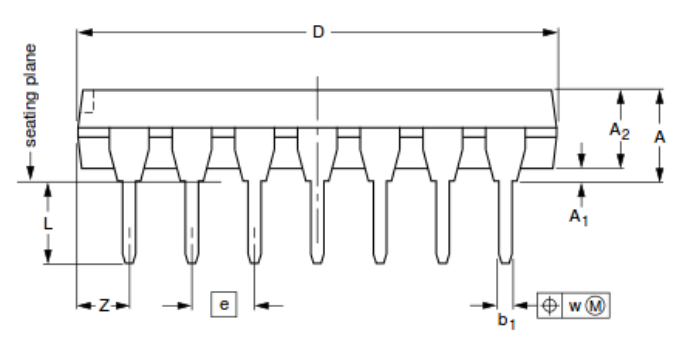
refined to



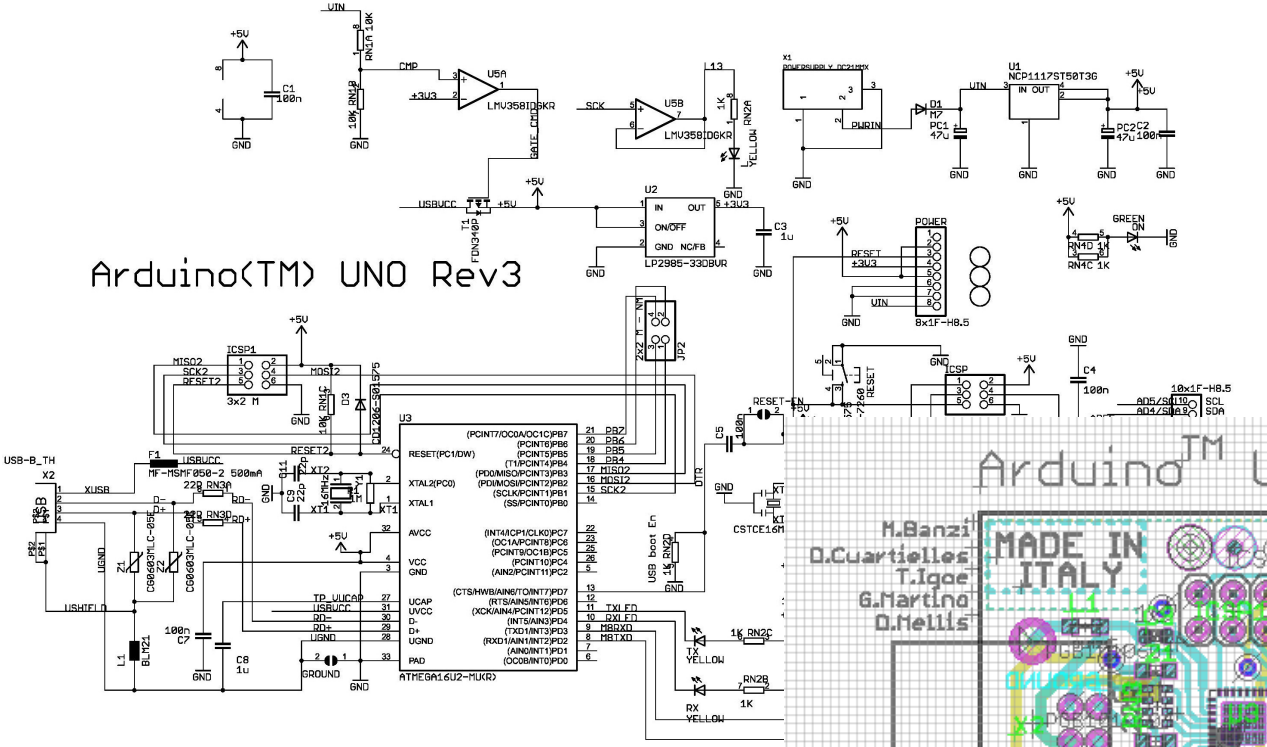
Aspect **Geometry**

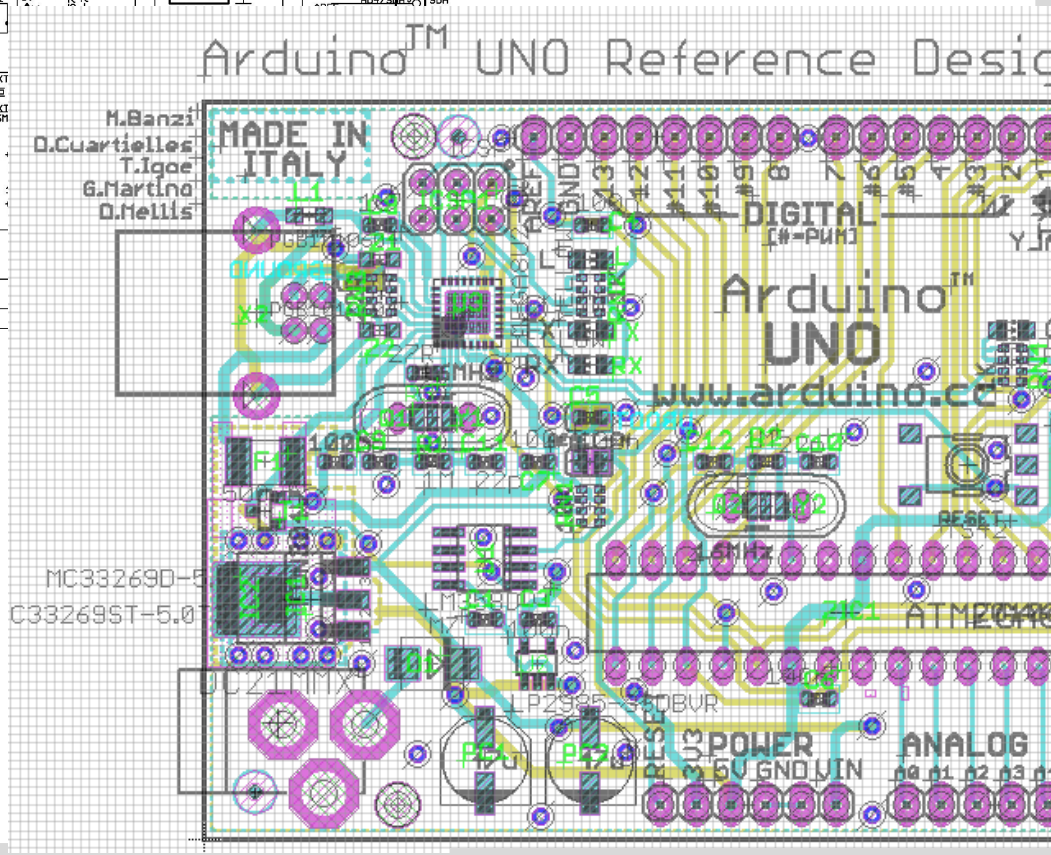


refined to

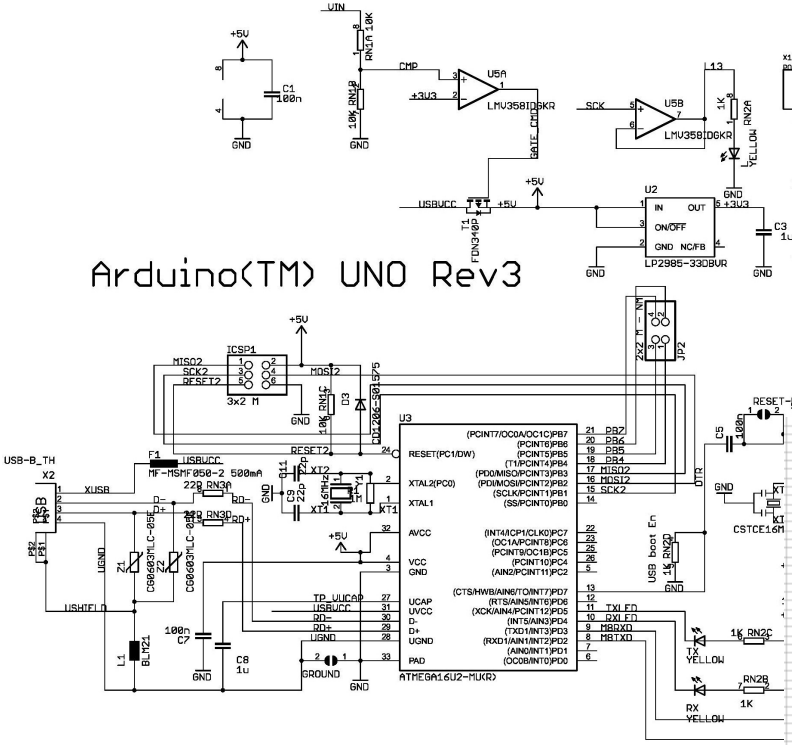


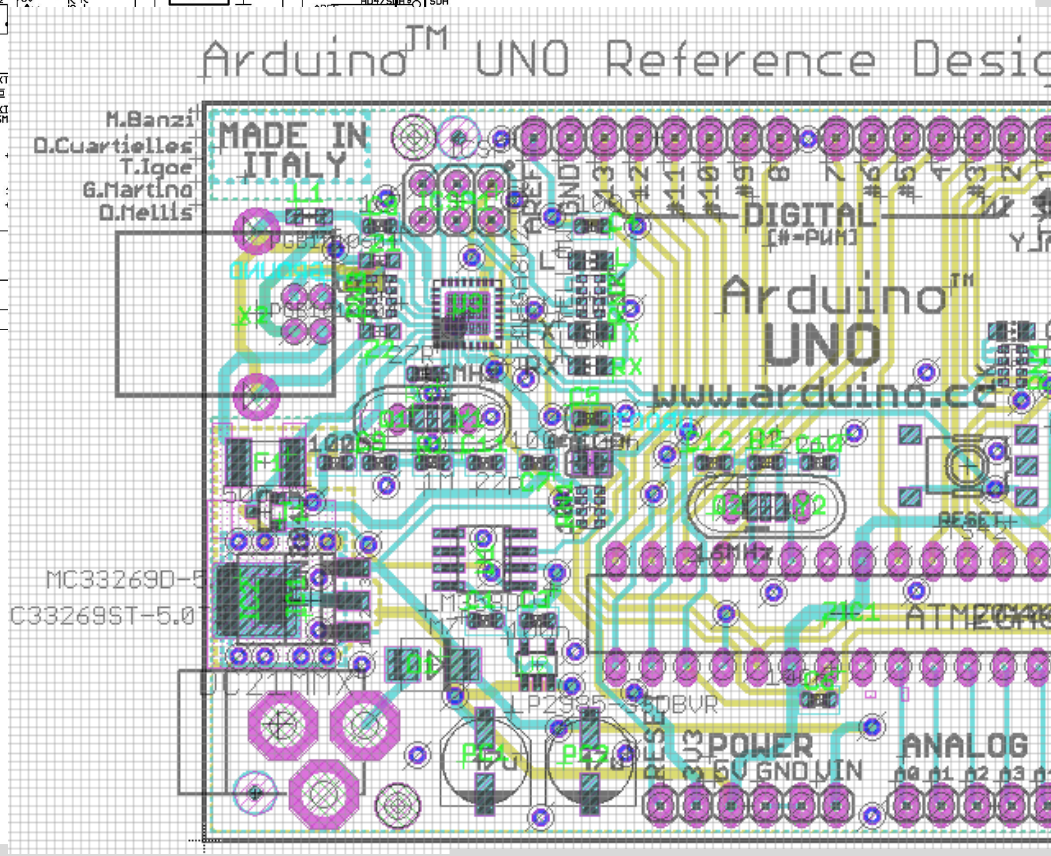
Schematic Diagram vs. PCB Layout





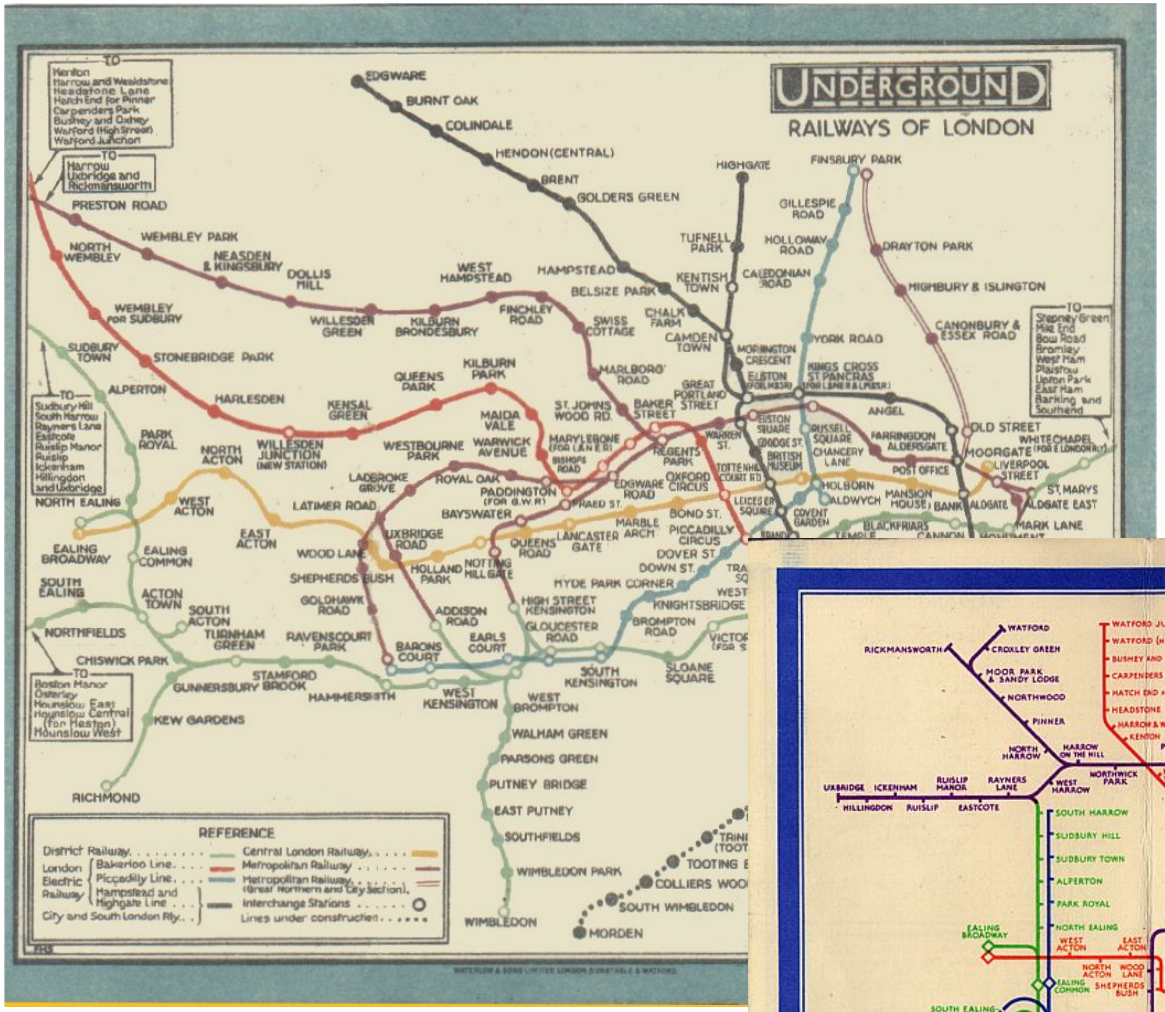
Schematic Diagram vs. PCB Layout



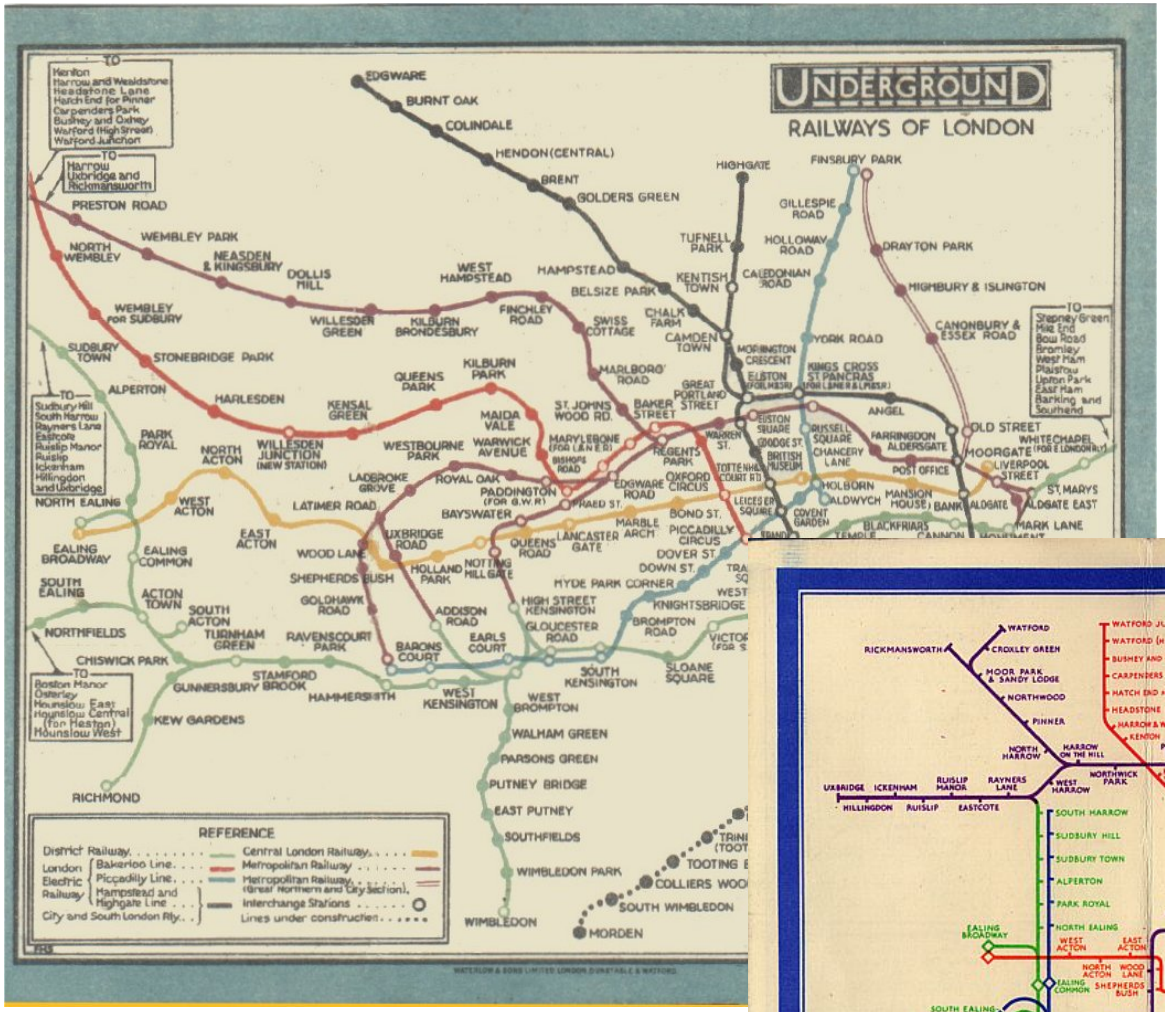


Aspect
“Behaviour”
preserved

Beck diagrams (1931)



Beck diagrams (1931)



Aspect
“Route planning”
is preserved

Property preservation

Abstraction with focus on particular aspect

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

Refinement with focus on particular aspect

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

Examples:

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

Property preservation

Abstraction with focus on particular aspect

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

Refinement with focus on particular aspect

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

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

Examples:

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

“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

Abstraction as a **technique**

“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:

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**

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

“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

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

Abstraction as a **technique**

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

“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

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

Abstraction as a **technique**

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

“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

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

Abstraction as a **technique**

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

“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

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

Abstraction as a **technique**

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

“Conceptual” vs “Technical” Abstraction

Two areas of abstraction and refinement in formal methods:

Conceptual abstraction

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

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

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)

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 **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

Contexts and Machines

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**:

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Contexts and Machines

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

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Contexts and Machines

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

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Contexts and Machines

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

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Contexts and Machines

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

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Contexts and Machines

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

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Contexts and Machines

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

Contexts

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

Machines

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

(Explanations or alternative names in parens)

Introduction by Example

Students and Exams – Requirements

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

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

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

➜ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.

Introduction by Example

Students and Exams – Requirements

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

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

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

➜ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.

Introduction by Example

Students and Exams – Requirements

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

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

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

➜ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.

Introduction by Example

Students and Exams – Requirements

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

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

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

➜ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.

Introduction by Example

Students and Exams – Requirements

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

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

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

➜ Identify the **context**, the **state** and the **events** according to the requirements R1–R3.

Introduction by Example

Students and Exams – Requirements

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.

Exam Context

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*

Exam Context

CONTEXT *ExamCtxt*

SETS

STUDENT // see requirement R1
SUBJECT

CONSTANTS

maths *physics* *siblings*

AXIOMS

$maths \in SUBJECT$ // type of variables
$physics \in SUBJECT$

Exam Context

CONTEXT *ExamCtxt*

SETS

STUDENT // see requirement R1
SUBJECT

CONSTANTS

maths *physics* *siblings*

AXIOMS

$maths \in SUBJECT$ // type of variables
$physics \in SUBJECT$
$maths \neq physics$ // constants could have same value

Exam Context

CONTEXT *ExamCtxt*

SETS

$STUDENT$ // see requirement R1
$SUBJECT$

CONSTANTS

$maths \quad physics \quad siblings$

AXIOMS

$maths \in SUBJECT$ // type of variables
$physics \in SUBJECT$
$maths \neq physics$ // constants could have same value
$siblings \subseteq STUDENT \times STUDENT$ // function type

Exam Context

CONTEXT *ExamCtxt*

SETS

$STUDENT$ // see requirement R1
$SUBJECT$

CONSTANTS

$maths \quad physics \quad siblings$

AXIOMS

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

Exam Machine

MACHINE *ExamAbstract*

Exam Machine

MACHINE *ExamAbstract*
SEES *ExamCtxt*

Exam Machine

MACHINE *ExamAbstract*
SEES *ExamCtxt*

VARIABLES
    passed *failed*

Exam Machine

MACHINE *ExamAbstract*
SEES *ExamCtxt*

VARIABLES
$passed \quad failed$
INVARIANTS
$passed \subseteq STUDENT \qquad failed \subseteq STUDENT$

Exam Machine

MACHINE *ExamAbstract*
SEES *ExamCtxt*

VARIABLES
$\quad passed \qquad failed$
INVARIANTS
$\quad passed \subseteq STUDENT \qquad failed \subseteq STUDENT$
$\quad passed \cap failed = \varnothing \qquad$ // R3

Exam Machine

MACHINE *ExamAbstract*
SEES *ExamCtxt*

VARIABLES
$passed \quad failed$
INVARIANTS
$passed \subseteq STUDENT \qquad failed \subseteq STUDENT$
$passed \cap failed = \varnothing$ // R3

EVENTS
INITIALISATION $\widehat{=} \ldots$
ATTEMPTEXAM $\widehat{=} \ldots$ // R2
PASSEXAM $\widehat{=} \ldots$ // R3
FAILEXAM $\widehat{=} \ldots$ // R3

Exam Machine (2)

MACHINE *ExamAbstract*
VARIABLES *passed failed* ...

EVENTS

INITIALISATION $\widehat{=}$

$$failed := \varnothing$$
$$passed := \varnothing$$

Exam Machine (2)

MACHINE *ExamAbstract*
VARIABLES *passed failed* ...

EVENTS

INITIALISATION $\widehat{=}$
$\quad$ $failed := \varnothing$
$\quad$ $passed := \varnothing$

PASSEXAM $\widehat{=}$
$\quad$ ANY $s\ grade$
$\quad$ WHERE $s \in STUDENT \wedge grade \leq 4$
$\quad$ THEN $passed := passed \cup \{s\}$

Exam Machine (2)

MACHINE *ExamAbstract*
VARIABLES *passed failed* ...

EVENTS

INITIALISATION $\widehat{=}$
$\quad$ $failed := \varnothing$
$\quad$ $passed := \varnothing$

PASSEXAM $\widehat{=}$
$\quad$ ANY $s\ grade$
$\quad$ WHERE $s \in STUDENT \wedge grade \leq 4$
$\quad$ THEN $passed := passed \cup \{s\}$

FAILEXAM $\widehat{=}$
$\quad$ ANY $s\ grade$
$\quad$ WHERE $s \in STUDENT \wedge grade > 4$
$\quad$ THEN $failed := failed \cup \{s\}$

Invariant violated

MACHINE *ExamAbstract*
VARIABLES *passed failed*
INVARIANTS $passed \cap failed = \varnothing \quad \ldots$

EVENTS
PASSEXAM $\widehat{=}$
  ANY *s grade*
  WHERE $s \in STUDENT \wedge grade \leq 4$
  THEN $passed := passed \cup \{s\}$

FAILEXAM $\widehat{=}$
  ANY *s grade*
  WHERE $s \in STUDENT \wedge grade > 4$
  THEN $failed := failed \cup \{s\}$

Invariant violated

MACHINE *ExamAbstract*
VARIABLES *passed failed*
INVARIANTS $passed \cap failed = \varnothing \quad \ldots$

EVENTS
PASSEXAM $\widehat{=}$
  ANY *s grade*
  WHERE $s \in STUDENT \setminus (failed \cup passed) \wedge grade \leq 4$
  THEN $passed := passed \cup \{s\}$

FAILEXAM $\widehat{=}$
  ANY *s grade*
  WHERE $s \in STUDENT \setminus (failed \cup passed) \wedge grade > 4$
  THEN $failed := failed \cup \{s\}$

Underspecified model

EVENTS

PASSEXAM $\widehat{=}$

ANY s $grade$ WHERE $grade \leq 4 \wedge s \in \ldots$

THEN $passed := passed \cup \{s\}$

FAILEXAM $\widehat{=}$

ANY s $grade$ WHERE $grade > 4 \wedge s \in \ldots$

THEN $failed := failed \cup \{s\}$

ATTEMPTEXAM $\widehat{=}$

ANY s $grade$ WHERE $grade \in \mathbb{N} \wedge s \in \ldots$

THEN $skip$

Underspecified model

EVENTS

PASSEXAM $\widehat{=}$
  ANY s $grade$ WHERE $grade \leq 4 \wedge s \in \ldots$
  THEN $passed := passed \cup \{s\}$

FAILEXAM $\widehat{=}$
  ANY s $grade$ WHERE $grade > 4 \wedge s \in \ldots$
  THEN $failed := failed \cup \{s\}$

ATTEMPTEXAM $\widehat{=}$
  ANY s $grade$ WHERE $grade \in \mathbb{N} \wedge s \in \ldots$
  THEN $skip$

Additional requirement

R4 Any student may attempt the exam three times and ultimately fails if the fourth attempt is unsuccessful.

Refinement Exams (1)

MACHINE *RefinedExams* REFINES *ExamsAbstract*

Refinement Exams (1)

MACHINE *RefinedExams* REFINES *ExamsAbstract*
VARIABLES *passed attempts*

Refinement Exams (1)

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

$attempts \in STUDENT \rightarrow \mathbb{N}$ // typing for *attempts*

$failed = \{s \cdot attempts(s) = 4\}$ // coupling invariant

Refinement Exams (1)

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

$attempts \in STUDENT \rightarrow \mathbb{N}$ // typing for *attempts*

$failed = \{s \cdot attempts(s) = 4\}$ // coupling invariant

EVENTS

INITIALISATION $\widehat{=}$ REFINES INITIALISATION

$passed := \varnothing$

$attempts := \{s \cdot s \in STUDENT \mid (s \mapsto 0)\}$

Refinement Exams (1)

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

$attempts \in STUDENT \rightarrow \mathbb{N}$ // typing for *attempts*

$failed = \{s \cdot attempts(s) = 4\}$ // coupling invariant

EVENTS

INITIALISATION $\widehat{=}$ REFINES INITIALISATION

$passed := \varnothing$

$attempts := \{s \cdot s \in STUDENT \mid (s \mapsto 0)\}$

EXAMULTIMATEFAIL $\widehat{=}$ REFINES EXAMFAIL ...
EXAMMISSED $\widehat{=}$ REFINES EXAMATTEMPT ...
EXAMPASSED $\widehat{=}$ REFINES EXAMPASSED ...

Refinement Exams (2)

...

EVENTS

EXAMULTIMATEFAIL $\widehat{=}$ REFINES EXAMFAIL

ANY s $grade$

WHERE ... $\wedge$ $grade > 4 \wedge attempts(s) = 3$

THEN

$attempts(s) := attempts(s) + 1$

EXAMMISSED $\widehat{=}$ REFINES EXAMATTEMPT

ANY s $grade$

WHERE ... $\wedge$ $grade > 4 \wedge attempts(s) < 3$

THEN

$attempts(s) := attempts(s) + 1$

...

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*

$\Rightarrow$ 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
- ...

Set Theory

Formal language in Event-B models

Typed First Order Set Theory with Additional Theories

Every term in Event-B has a unqiue type.

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

$$maths \in failed \quad \text{is syntactially invalid.}$$

(remember: $math \in SUBJECT$, $failed \subseteq STUDENT$)

"You can't compare apples and oranges."

Set Theory

Formal language in Event-B models

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

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

Set Theory

Formal language in Event-B models

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

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

Set Theory

Formal language in Event-B models

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

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

Set Theory

Formal language in Event-B models

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

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

Set Theory

Formal language in Event-B models

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

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

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 $\mathbb{P}(T)$ 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 produt).

Every expression E has a unqiue 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 produt).

Every expression E has a unqiue 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 produt).

Every expression E has a unqiue 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 produt).

Every expression E has a unqiue 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 produt).

Every expression E has a unqiue type $\tau(E)$.

Types (2)

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

Reasons to introduce types:

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

Types (2)

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

Reasons to introduce types:

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

Russell's paradox

Assume that the expression $\{s \mid \phi\}$ for any formula ϕ denotes a set. Let $R := \{s \mid s \notin s\}$.

Types (2)

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

Reasons to introduce types:

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

Russell's paradox

Assume that the expression $\{s \mid \phi\}$ for any formula ϕ denotes a set. Let $R := \{s \mid s \notin s\}$.
One observes: $R \in R \iff R \notin R \quad \lightning$

Types (2)

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

Reasons to introduce types:

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

Russell's paradox

Assume that the expression $\{s \mid \phi\}$ for any formula ϕ denotes a set. Let $R := \{s \mid s \notin s\}$. Not allowed with types.
One observes: $R \in R \iff R \notin R \quad \lightning$
(This crushed naive set theory in early 1900s.)

Sets

Constructors for sets:

- empty set $\varnothing : \mathbb{P}(S)$

Sets

Constructors for sets:

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

Sets

Constructors for sets:

- empty set $\varnothing : \mathbb{P}(S)$
- set extension $\{\ldots\} : S^* \to \mathbb{P}(S)$
  example: $\{1, 2\} : \mathbb{P}(\mathbb{Z})$
- carrier sets $C : \mathbb{P}(C)$
  example: $STUDENT : \mathbb{P}(STUDENT)$

Sets

Constructors for sets:

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

Sets

Constructors for sets:

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

Sets

Constructors for sets:

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

Relations

- Relations are sets of pairs (tuples).

Relations

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

Relations

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

Relations

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

Relations

- Relations are sets of pairs (tuples).
- All relations: $E_1 \leftrightarrow E_2 := \mathbb{P}(E_1 \times E_2)$
- Pairs $(E_1 \mapsto E_2) : \tau(E_1) \times \tau(E_2)$
- Domain of a relation $dom(R)$
  $\mathrm{dom}(R) = \{x, y \cdot (x \mapsto y) \in R \mid x\}$
  example: $\mathrm{dom}(E_1 \times E_2) = E_1$
- Range of a relation $ran(R)$
  $\mathrm{ran}(R) = \{x, y \cdot (x \mapsto y) \in R \mid y\}$
  example: $\mathrm{ran}(E_1 \times E_2) = E_2$

Relations

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