Event-B Course

1. Introduction

Jean-Raymond Abrial

September-October-November 2011

Purpose of the Course (1)

- To show that software (and systems) can be correct by construction

- Insights about modeling and formal reasoning using Event-B

- To show that this can be made practical with the Rodin Platform

Purpose of the Course (2)

- To illustrate this approach with many examples:

  - a small sequential program

  - controlling cars on a bridge

  - a mechanical press controller

  - a file transfer protocol

Purpose of the Course (3)

- More examples:

  - a mobile phone routing algorithm

  - more sequential programs

  - some hardware developments

  - an access control system

  - . . .

What you Will Learn (1)

- Writing a requirement document (more explanations later)
- Modeling versus programming (more explanations later)
- Abstraction and refinement (more explanations later)

What you Will Learn (2)

- Some mathematical techniques used for reasoning

- The practice of proving as a means to construct programs

- The usage of the Rodin Platform

Lectures: Monday (10:10 to 12:00) and Wednesday (13:00 to 14:50)

Practices: Tuesday (18:40 to 20:30)

17 lectures:

- September: 5, 7, 14, 19, 21, 26, 28
- October: 10, 12, 17, 19, 24, 26, 31
- November: 2, 7, 9

9 Practices:

- September: 6, 13, 20, 27
- October: 11, 18, 25
- November: 1, 8

Provisional Schedule for the 17 Lectures

- 1. Introduction (September 5, 7)
- 2. Cars on a Bridge (September 14, 19, 21)
- 3. Mechanical Press (September 26, 28)
- 4. File Transfer Protocol (October 10)
- 5. Math Refresher (October 12, 17)
- 6. Mobile Phone Routing (October 19)
- 7. Hardware Development (October 24, 26)
- 8. Access Control System (October 31, November 2)
- 9. Hypervisor Development (November 7, 9)

Provisional Schedule for the 9 Practices

- 1. Writing a Requirement Document (September 6)
- 2. Introducing the Rodin Platform (September 13)
- 3. Developing a small Motor Controller (September 20)
- 4. Practicing Interactive Proofs (September 27, October 11, 18)
- 5. Another Formal Development (October 25)
- 6. Developing a Business Protocol (November 1, 8)

Material to be Distributed

For lectures:

- slides
- text

For practices:

- text of exercises
- corrected exercises (one week later)
- Rodin Platform development files

Outline of this Introductory Lecture

1. About formal methods in general

2. About requirements

3. About modeling

4. A light introduction to Event-B

5. Presentation of a small example

1. About formal methods in general

Formal Methods: a Great Confusion

- What are they used for?

- When are they to be used?

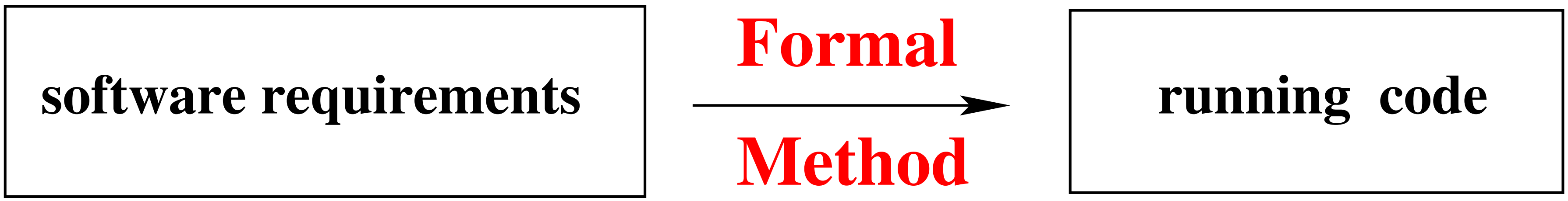
- Is UML a formal method?

- Are formal methods needed when doing OO programming?

- What is their definition?

Purpose of Formal Methods

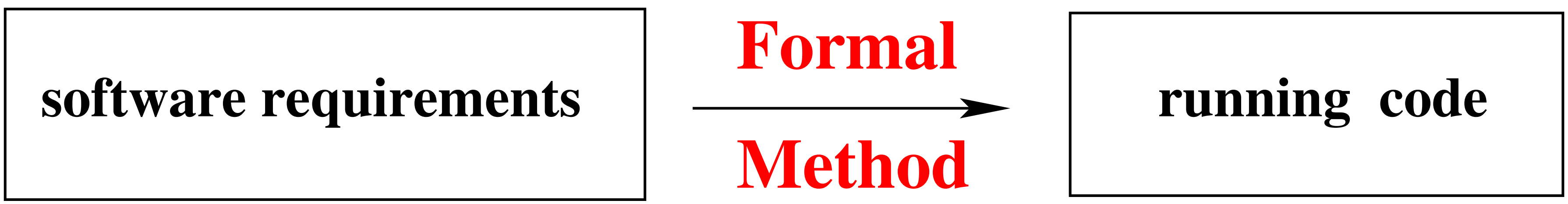
- Helping people in doing the following transformation:



- It does not seem to be different from ordinary programming

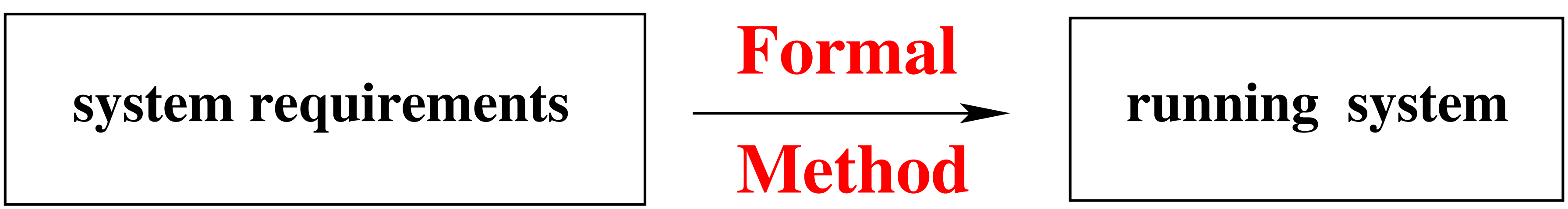
Purpose of Formal Methods

- Helping people in doing the following transformation:



- It does not seem to be different from ordinary programming

- It can be generalized to:



Preliminary Definitions and Categories

- Determining whether a program has certain wishful properties.

- The checked properties will become clearer in subsequent slides

- Different kinds of formal methods (according to this definition)
    - Type checking
    - Abstract interpretation
    - Model checking
    - Theorem proving

Type Checking (1)

- The properties to be checked are properties of program variables

- Controlling low level properties of variables

- A type defines:
    - a set of values to be assigned to a variable
    - the operations that can be performed on a variable
    - the way a program variable will be stored in the memory

Type Checking (2)

- Type checking controls that:
    - value assignments to a variable is correct
    - the variable is used in authorized operations only

- It is done automatically by compilers

Abstract Interpretation (1)

- The property to be checked is the absence of run-time errors

- Typical run-time detected:
  - Division by zero
  - Array bound overflow
  - Arithmetic overflow (floating point)

Abstract Interpretation (2)

- The analysis is performed by abstracting the program variables

- Executing the resulting abstraction rather than the program itself

- Once the property is defined, it is an automatic technique

- Models to be studied usually denote finite state machines

- Properties to be checked:
    - Reachability
    - Deadlock freeness

- Once the property is defined, it is an automatic technique

Theorem Proving: Model Construction (1)

- Properties to be checked are any of the above

- But more abstract properties can also be checked (more later)

- This is the approach developed in this course

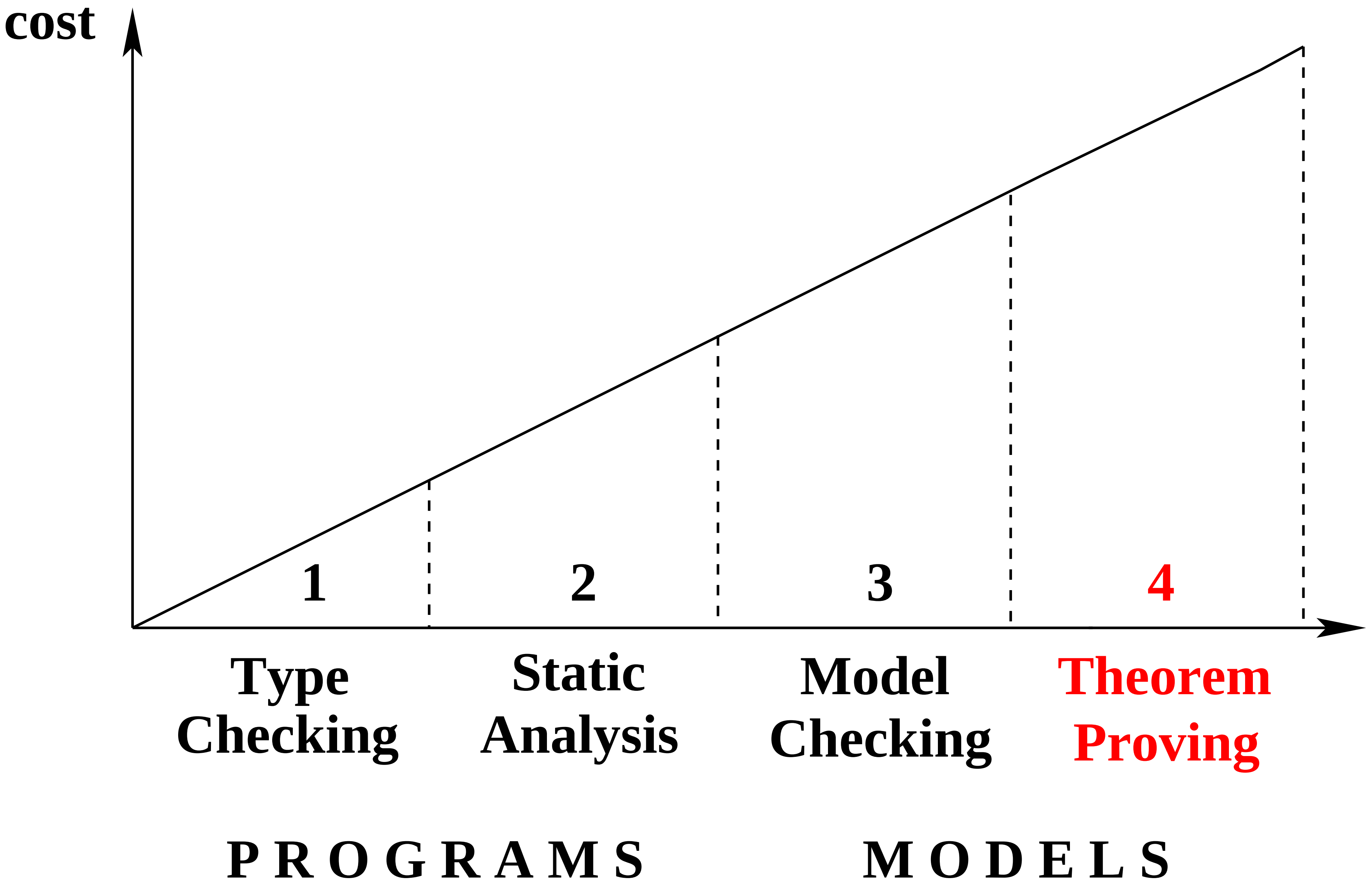
Theorem Proving: Model Construction (2)

- One constructs models by successive refinements

- The properties to be proved are parts of the models

- The most refined model is automatically translated into a program

Comparison (1)



Comparison (2)

	Nature	Properties
type checking	programs	defined within the program
abstract interpretation	programs	defined after writing program
model checking	models	defined after writing model
theorem proving	models	defined within the model

Why Using Formal Methods with Proofs?

- When the risk is too high (e.g. in embedded systems).

- When the verifications of other approaches are not sufficient

- When people question their industrial development process.

- Decision of using formal methods is always strategic.

How about other Engineering Disciplines?

How about other Engineering Disciplines?

How about other Engineering Disciplines?

Other Disciplines

- Some mature disciplines:
  - Avionics,
  - Civil Engineering,
  - Mechanical Constructions,
  - Ship building,
  - . . .

- Some mature disciplines:
    - Avionics,
    - Civil Engineering,
    - Mechanical Constructions,
    - Ship building,
    - . . .

- Does there exist methods similar to formal methods ?

- Some mature disciplines:
  - Avionics,
  - Civil Engineering,
  - Mechanical Constructions,
  - Ship building,
  - . . .

- Does there exist methods similar to formal methods ?

- Yes

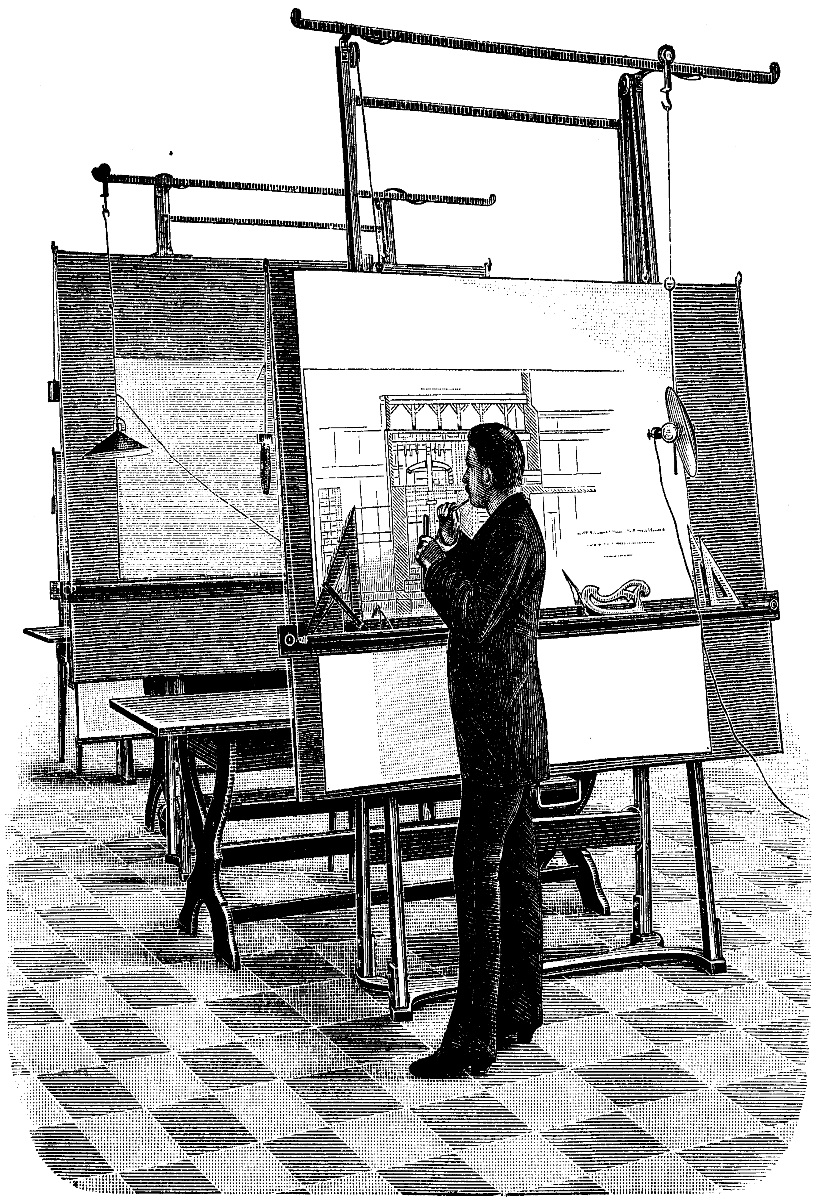
Other Disciplines

- Some mature disciplines:
  - Avionics,
  - Civil Engineering,
  - Mechanical Constructions,
  - Ship building,
  - . . .

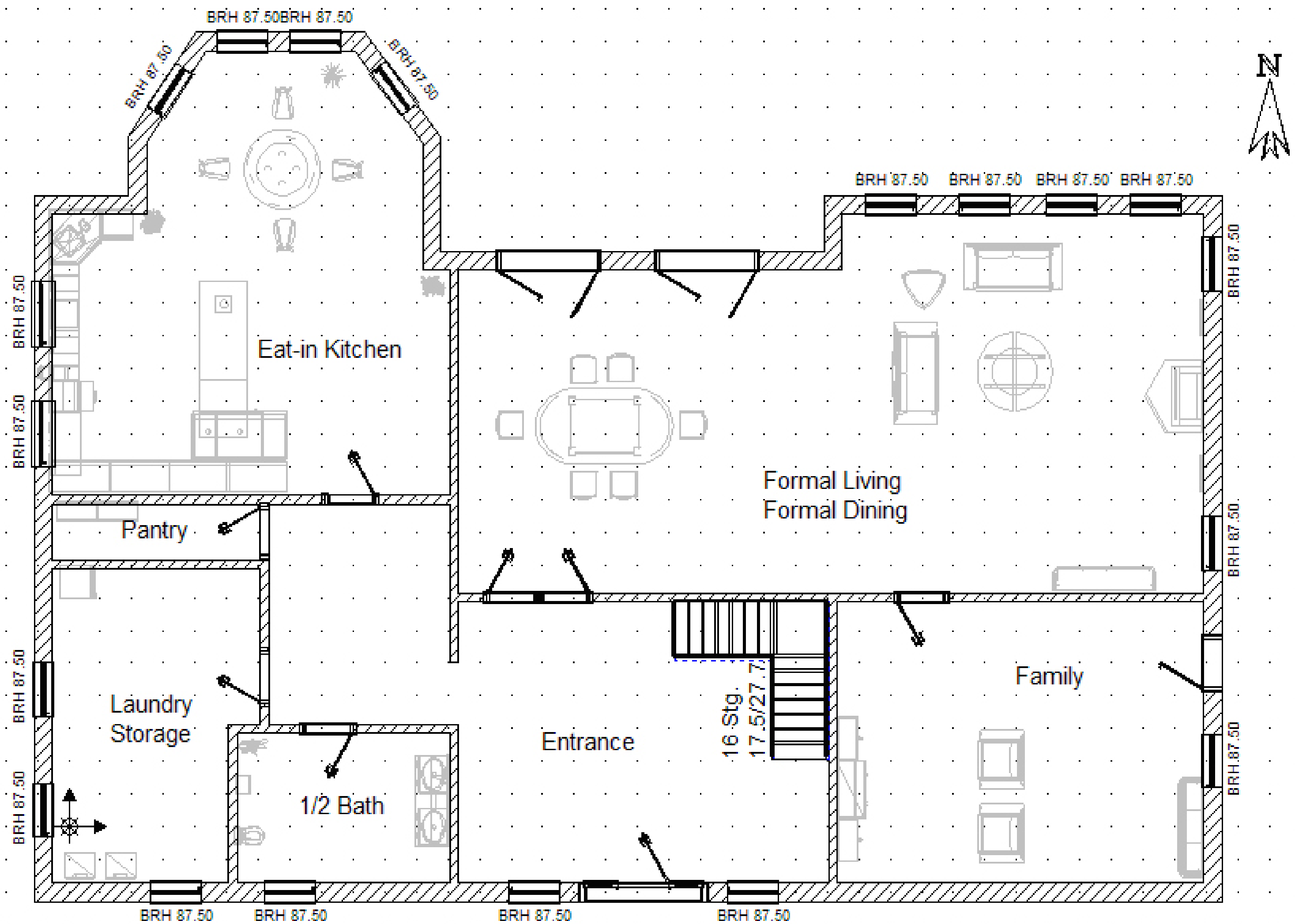
- Does there exist methods similar to formal methods ?

- Yes, Blueprints

Blueprints



Architect Blueprint



What is a Blue Print?

- An abstract representation of the system we want to build

- The basis is lacking (you cannot “drive” the blue print of a car)

- Allows to reason about the system during its design, NOT AFTER

- Example: constructing a freeway or a bridge

What is a Blue Print?

- An abstract representation of the system we want to build

- The basis is lacking (you cannot “drive” the blue print of a car)

- Allows to reason about the system during its design, NOT AFTER

- Example: constructing a freeway or a bridge

- Is it important? (according to professionals)

What is a Blue Print?

- An abstract representation of the system we want to build

- The basis is lacking (you cannot “drive” the blue print of a car)

- Allows to reason about the system during its design, NOT AFTER

- Example: constructing a freeway or a bridge

- Is it important? (according to professionals) YES

Reasoning about the Intended System?

- Defining and calculating its behavior (what it does)

- Incorporating constraints (what it must not do)

- Defining architecture

- Based on some underlying theories
    - strength of materials,
    - fluid mechanics,
    - gravitation,
    - etc.

Techniques of “Blue Printing”

- Using pre-defined conventions (often computerized these days)

- Conventions should help facilitate reasoning

- Adding details on more accurate versions

- Postponing choices by having some open options

- Decomposing one blue print into several

- Reusing “old” blue prints (with slight changes)

2. About requirements

Requirement Document Outline

- Place of requirement document in the system life cycle

- Difficulties and weak point

- Characterizing the requirement document

- Proposing some structuring rules

1. Feasibility Study
2. Requirement Analysis
3. Technical Specification
4. Design

4. Coding
5. Test
6. Documentation
7. Maintenance

Difficulties and Weak Points

- Ensuring relative consistency between the phases

- Formal Methods could help (in the later phases)

- But still a problem in the earlier phases

- Weakest part: the requirement document

About the Requirement Document

- Importance of this document (due to its position in the life cycle)

- Obtaining a good requirement document is not easy:
  - missing points
  - too specific (over-specified)

- Industrial requirement document are usually difficult to exploit

About the Requirement Document (cont'd)

- Hence very often necessary to rewrite it

- It will cost a significant amount of time and money (but well spent)

- The famous specification change syndrome might disappear

Some Structuring Rules

- Two separate texts in the same document:
  - explanatory text: the why
  - reference text: the what

- Reference text (what) and explanatory text (why) defined together

- The reference text eventually becomes the official document

- Must be signed by concerned parties

The Reference Text (1)

- Contains the properties of the future system

- Contains the assumptions about its environment

- The properties must hold for the system to be correct

- This must be the case if the assumptions hold

The Reference Text (2)

- It is made of short labeled English statements

- Should be easy to read (different font) and easy to extract (boxed)

- The problem of the traceability

An Example of Requirement Document (1)

- We show the embedding of the Explanations and the References

- Explanation:

- The function of this system is to control cars on a narrow bridge.

- This bridge is supposed to link the mainland to a small island.

Kinds of Requirements (labels)

- There are two kinds of requirements:

    - the equipment (environment) labeled EQP,

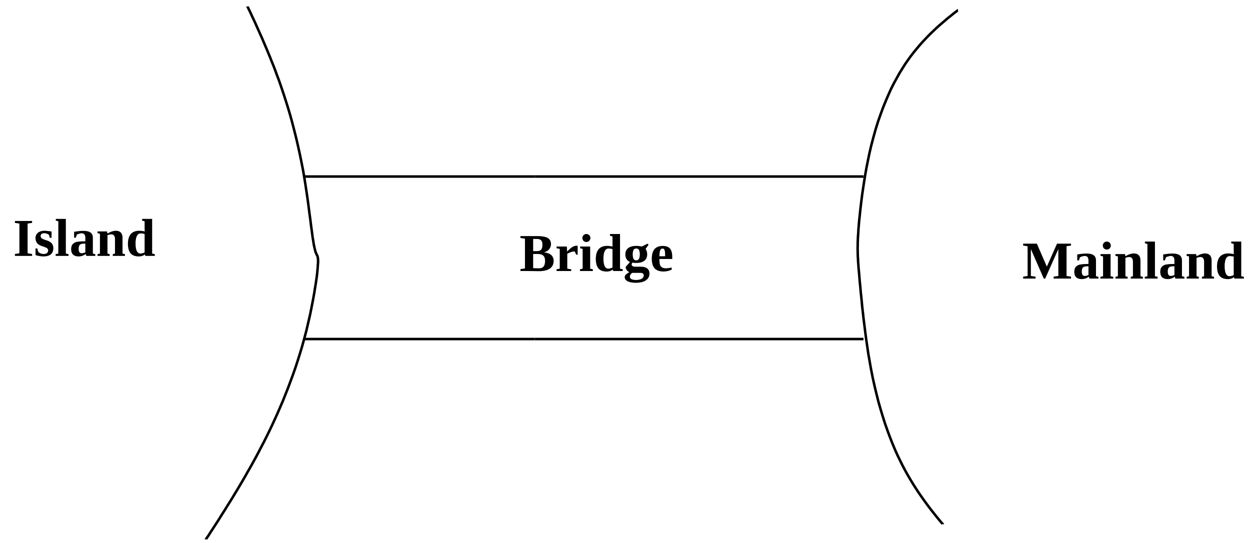
    - the function of the system, labeled FUN.

An Example of Requirement Document (2)

- Reference:

The system is controlling cars on a bridge between the mainland and an island	FUN-1

- Explanation: This can be illustrated as follows



An Example of Requirements Document (3)

- Explanation: The controller is equipped with two traffic lights.

- Reference:

The system has two traffic lights with two colors: green and red	EQP-1

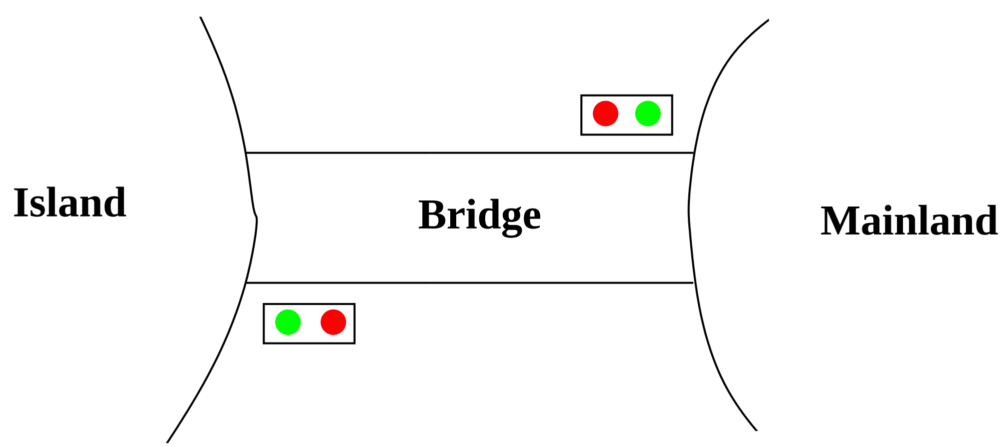
An Example of Requirements Document (4)

- Explanation:

- One of the traffic lights is situated on the mainland.

- The other one on the island.

- This can be illustrated as follows:



An Example of Requirements Document (5)

- Reference:

The traffic lights control the entrance to the bridge at both ends of it	EQP-2

- Explanation: Drivers are supposed to obey the traffic light

- Reference:

Cars are not supposed to pass on a red traffic light, only on a green one	EQP-3

An Example of Requirements Document (6)

- Explanation:

- There are also four car sensors

- These sensors are situated at both ends of the bridge.

- They are supposed to detect the presence of cars

- Reference:

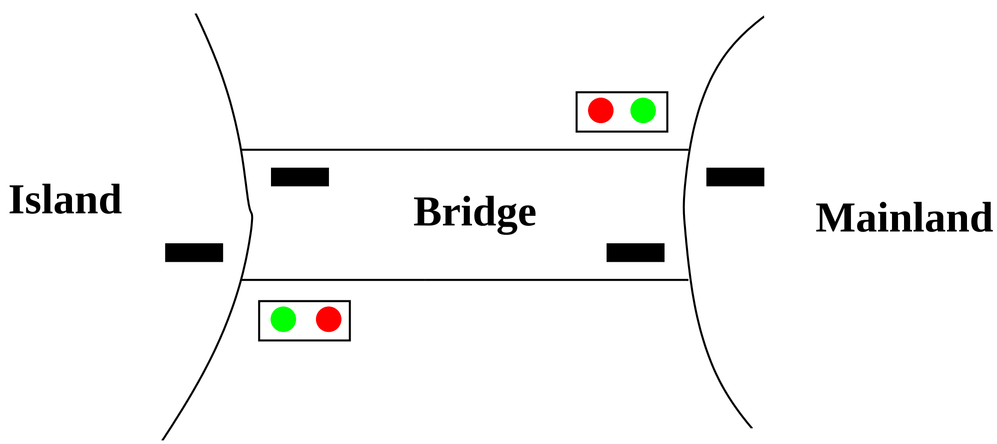
The system is equipped with four car sensors each with two states: on or off	EQP-4

An Example of Requirements Document (7)

- Reference:

The sensors are used to detect the presence of cars entering or leaving the bridge	EQP-5

- Explanation: The pieces of equipment can be illustrated as follows:



An Example of Requirements Document (8)

- Explanation: This system has two main constraints:

    - the number of cars on the bridge and the island is limited

    - the bridge is one way.

An Example of Requirements Document (9)

- Reference:

The number of cars on the bridge and the island is limited	FUN-2

The bridge is one way or the other, not both at the same time	FUN-3

Summary of the References (1)

The system is controlling cars on a bridge between the mainland and an island	FUN-1

The number of cars on the bridge and the island is limited	FUN-2

The bridge is one way or the other, not both at the same time	FUN-3

Summary of the References (2)

The system has two traffic lights with two colors: green and red	EQP-1

The traffic lights control the entrance to the bridge at both ends of it	EQP-2

Cars are not supposed to pass on a red traffic light, only on a green one	EQP-3

Summary of the References (3)

The system is equipped with four car sensors each with two states: on or off	EQP-4

The sensors are used to detect the presence of cars entering or leaving the bridge	EQP-5

3. About modeling

Back to the Questions about Formal Methods

- What are they used for?

- When are they to be used?

- Is UML a formal method?

- Are formal methods needed when doing OO programming?

- What is their definition?

Definitions of Formal Methods Used in this Course

- Formal methods are techniques for building and studying blue prints

- Such blue prints should be ADAPTED TO OUR DISCIPLINE

- Our discipline: design of hardware and software SYSTEMS

- These blue prints are now called models

- Reminder:
    - Models allow to reason about a FUTURE system
    - The basis is lacking (hence you cannot “execute” a model)

Reminders About Blue Prints (Models)

- Models allow to reason about a FUTURE system

- The basis is lacking (hence you cannot “execute” a model)

- Using pre-defined conventions

- Conventions should help facilitate reasoning (more to come)

Conventions for Model Writing and Reasoning

- Using ordinary discrete mathematical conventions

- Classical Logic (Predicate Calculus)

- Basic Set Theory (sets, relations and functions)

- Such conventions will be reviewed in subsequent lectures

Systems we are Interested to Develop (1)

- a “classical” piece of software

- an electronic circuit

- a file transfer protocol

- an airline booking system

- a PC operating system

Systems we are Interested to Develop (2)

- a nuclear plant controller

- a Smart-Card electronic purse

- a launch vehicle flight controller

- a driverless train controller

- a mechanical press controller

- etc.

Characterizing such Systems (1)

- They are made of many parts

- They interact with a possibly hostile environment

- They involve several executing agents

Characterizing such Systems (2)

- They require a high degree of correctness

- There construction spreads over several years

- Their specifications are subjected to many changes

Characterizing such Systems (3)

- These systems operate in a discrete fashion

- Their dynamical behavior can be abstracted by:

    - A succession of steady states

    - Intermixed with sudden jumps

Characterizing such Systems (4)

- The possible number of state changes are enormous

- Usually such systems never halt

- They are called DISCRETE TRANSITION SYSTEMS

- Test reasoning (a vast majority): VERIFICATION

- Blue Print reasoning (a very few): CORRECT CONSTRUCTION

Test Reasoning: VERIFICATION

- Based on laboratory execution
- Obvious incompleteness
- The oracle is usually missing
- Properties to be checked are chosen a posteriori
- Re-adapting and re-shaping after testing
- Reveals an immature technology

Blue Print Reasoning: CORRECT CONSTRUCTION

- Based on a formal model: the "blue print"

- Gradually describing the system with the needed precision

- Relevant Properties are chosen a priori

- Serious thinking made on the model, not on the final system

- Reasoning is validated by proofs

- Reveals a mature technology

Blue Print Reasoning: Outcome of Proving

- The proof succeeds

- The proof fails but refutes the statement to prove
    - the model is erroneous: it has to be modified

- The proof fails but is probably provable
    - the model is badly structured: it has to be reorganized

- The proof fails and is probably not provable nor refutable
    - the model is too poor: it has to be enriched

Price of Interactive Proofs in Industrial Projects

- n: number of lines of generated code

- n: number of lines of generated code

- f: proof factor. Typical values are 2 or 3.

  n/f is the number of proofs generated

- n: number of lines of generated code

- f: proof factor. Typical values are 2 or 3.

  n/f is the number of proofs generated

- x: percentage of interactive proofs. Typical values are 2, 5, 10.

  $n.x/100.f$ is the number of interactive proofs generated

- n: number of lines of generated code

- f: proof factor. Typical values are 2 or 3.

  n/f is the number of proofs generated

- x: percentage of interactive proofs. Typical values are 2, 5, 10.

  $n.x/100.f$ is the number of interactive proofs generated

- p: number of interactive proofs per man-day. Typical value is 20.

  $n.x/100.f.p$ is the number of man-day for the interactive proofs

Price of Interactive Proofs in Industrial Projects

- n: number of lines of generated code
- f: proof factor. Typical values are 2 or 3.

  n/f is the number of proofs generated
- x: percentage of interactive proofs. Typical values are 2, 5, 10.

  $n.x/100.f$ is the number of interactive proofs generated
- p: number of interactive proofs per man-day. Typical value is 20.

  $n.x/100.f.p$ is the number of man-day for the interactive proofs
- m: number of man-months to perform the interactive proofs.

  $n.x/100.f.p.20$ is the number of man-month for proving

- $m = n.x/100.f.p.20$ is the number of man-months needed for proving

n	$100,000$	$100,000$	$100,000$
f	2	2	2
x	2.5%	5%	10%
p	20	20	20
m	3.12	6.25	12.5

This shows the importance to prove as many automatic proofs as we can

About Formal Proofs in Industry (Some Figures)

- Rules of Thumb:

> n lines of final code implies $n/3$ proofs
>
> 95% of proofs discharged automatically
>
> 5% of proofs discharged interactively
>
> 350 interactive proofs per man-month

- 60,000 lines of final code $\rightsquigarrow$ 20,000 proofs $\rightsquigarrow$ 1,000 int. proofs
- 1,000 interactive proofs $\rightsquigarrow$ 1000/350 $\simeq$ 3 man-months
- Far less expensive than heavy testing

4. A Light Introduction to Event-B

- Event-B is not a programming language (even very abstract)

- Event-B is a notation used for developing mathematical models

- Mathematical models of discrete transition systems

- http://www.event-b.org

- Such models, once finished, can be used to eventually construct:
    - sequential programs,
    - distributed programs,
    - concurrent programs,
    - electronic circuits,
    - large systems involving a possibly fragile environment,
    - . . .

- The underlined statement is an important case.

- In this lecture, we shall construct a small sequential programs.

Action Systems developed by the Finnish school (Turku):

R.J.R. Back and R. Kurki-Suonio

Decentralization of Process Nets with Centralized Control.

2nd ACM SIGACT-SIGOPS Symposium

Principles of Distributed Computing (1983)

M.J. Butler

Stepwise Refinement of Communicating Systems.

Science of Computer Programming (1996)

The State of a Model

- A discrete model is first made of a state

- The state is represented by some constants and variables

- Constants are linked by some axioms

- Variables are linked by some invariants

- Axioms and invariants are written using set-theoretic expressions

The Events of a Model (Transitions)

- A discrete model is also made of a number of events

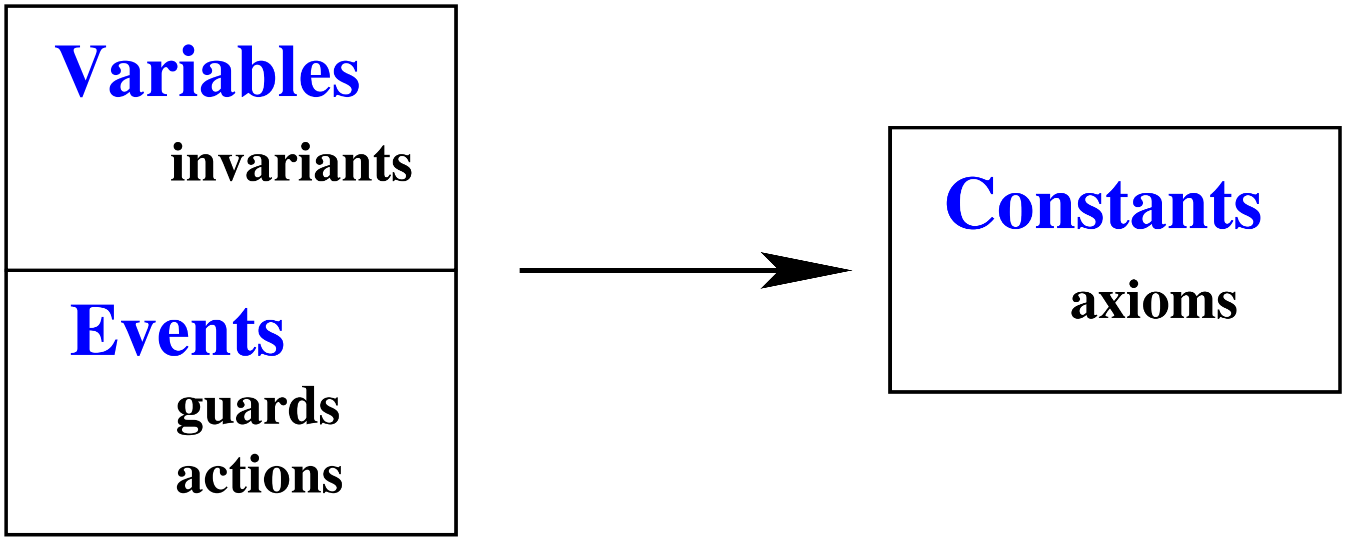
- An event is made of a guard and an action

- The guard denotes the enabling condition of the event

- The action denotes the way the state is modified by the event

- Guards and actions are written using set-theoretic expressions

A Model Schematic View



Dynamic Parts

(Machines)

Static Parts

(Contexts)

- An event execution is supposed to take no time

- Thus, no two events can occur simultaneously

- When all events have false guards, the discrete system stops

- When some events have true guards, one of them is chosen non-deterministically and its action modifies the state

- The previous phase is repeated (if possible)

```
Initialize;
while (some events have true guards) {
   Choose one such event;
   Modify the state accordingly;
}
```

Comments on the Operational Interpretation

- Stopping is not necessary: a discrete system may run for ever

- This interpretation is just given here for informal understanding

- The meaning of such a discrete system will be given by the proofs which can be performed on it.

Being more Precise: Machines and Contexts

- A model is made of several components

- A component is either a machine or a context:

Machine

- variables
- invariants
- theorems
- events

Context

- carrier sets
- constants
- axioms
- theorems

Machines and Contexts (cont'd)

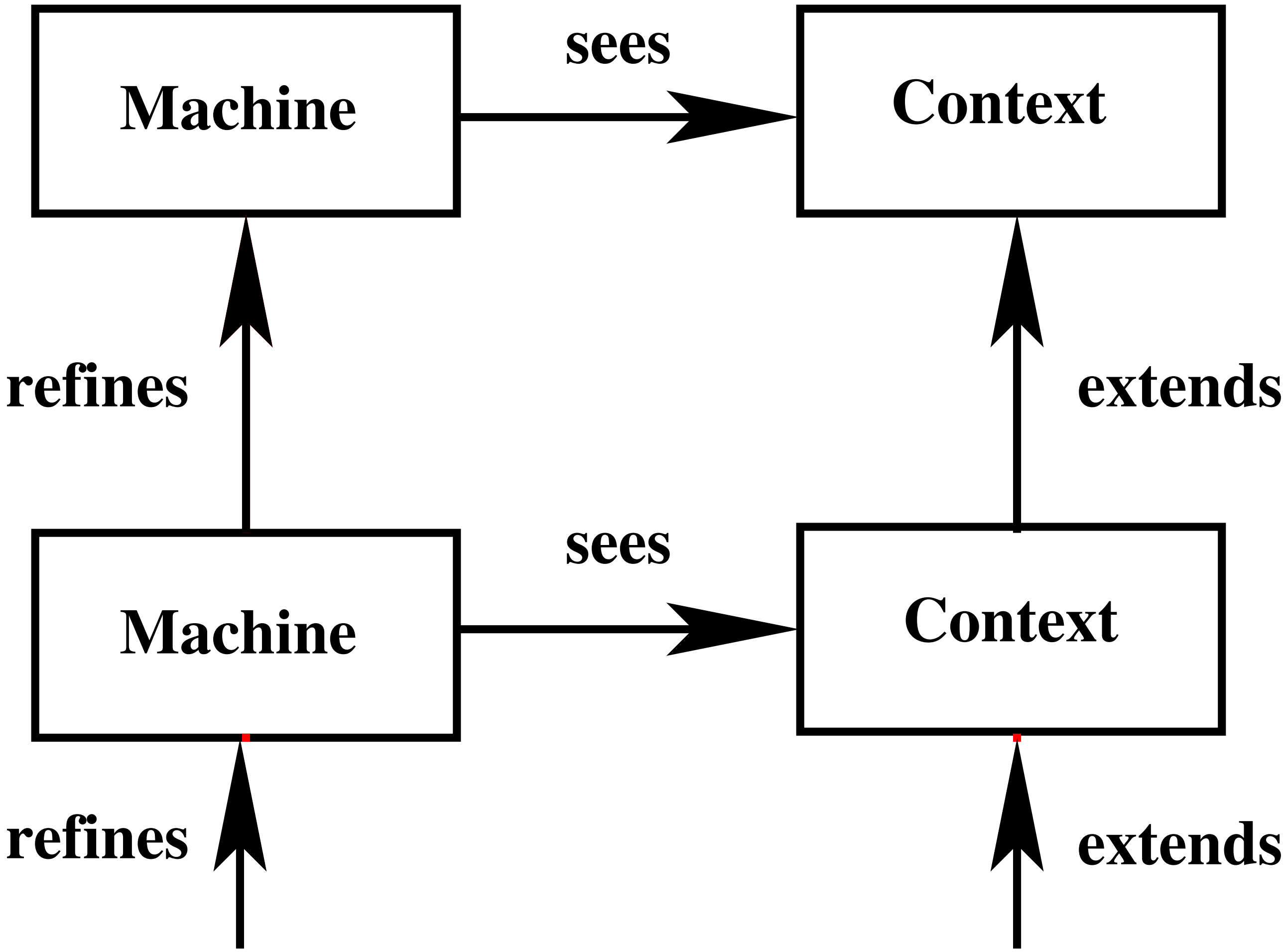
- Contexts contain the static structure of a discrete system (constants and axioms)

- Machines contain the dynamic structure of a discrete system (variables, invariants, and events)

- Machines see contexts

- Contexts can be extended

- Machines can be refined


sees
Machine
Context
refines
extends
sees
Machine
Context
refines
extends

5. Presentation of a Small Example

We are given a non-empty finite array of natural numbers	FUN-1

We like to find the maximum of the range of this array	FUN-2

Requirements for a First Small Example

We are given a non-empty finite array of natural numbers	FUN-1
We like to find the maximum of the range of this array	FUN-2

We want to find that 10 is the greatest element of this array

9	3	10	8	3	5

Development Strategy

- First, we show an initial model specifying the problem

- Later, we refine our model to produce an algorithm.

- In the initial model, we have:

    - a context where the constant array is defined

    - a machine where the maximum is "computed"

- Constant n denotes the size of the non-empty array,

- Constant f denotes the array,

- Constant M denotes a natural number.

constants:	n f M

$$0 < n$$

$$f \in 1 \,..\, n \rightarrow 0 \,..\, M$$

$$\mathrm{ran}(f) \neq \varnothing$$

- Mind the inference typing

Initial Model: the Context as presented in the slides

- Constant n denotes the size of the non-empty array,

- Constant f denotes the array,

- Constant M denotes a natural number.

constants: n
f
M

axm0_1: $0 < n$

axm0_2: $f \in 1 \,..\, n \rightarrow 0 \,..\, M$

thm0_1: $\mathrm{ran}(f) \neq \varnothing$

- Mind the inference typing

Context
sets
constants
axioms
theorems

Notice that we have no set

context
maxi_ctx_0
constants
n
f
M
axioms

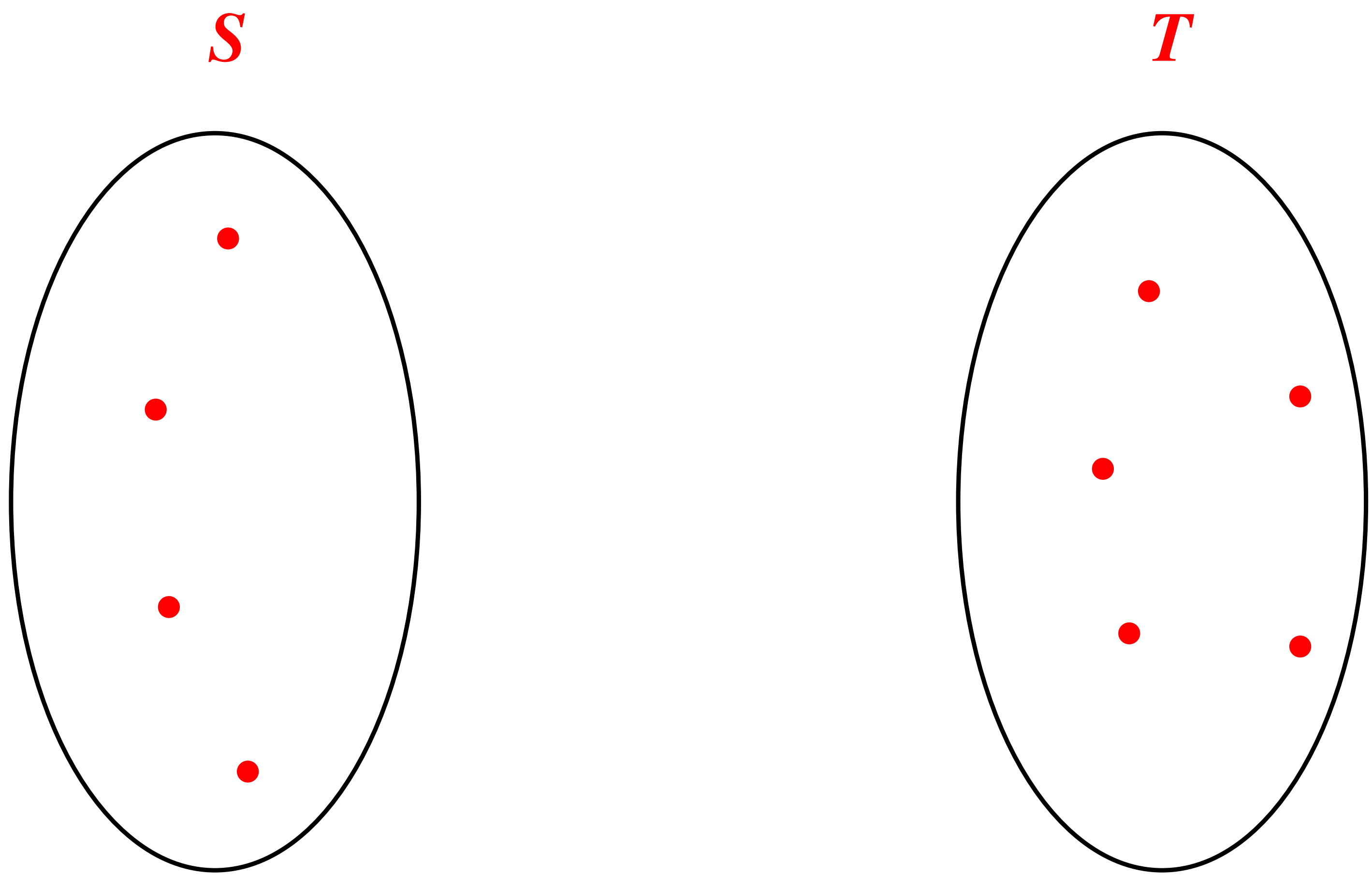
axm1 : $0 < n$

axm2 : $f \in 1..n \to 0\,..\,M$

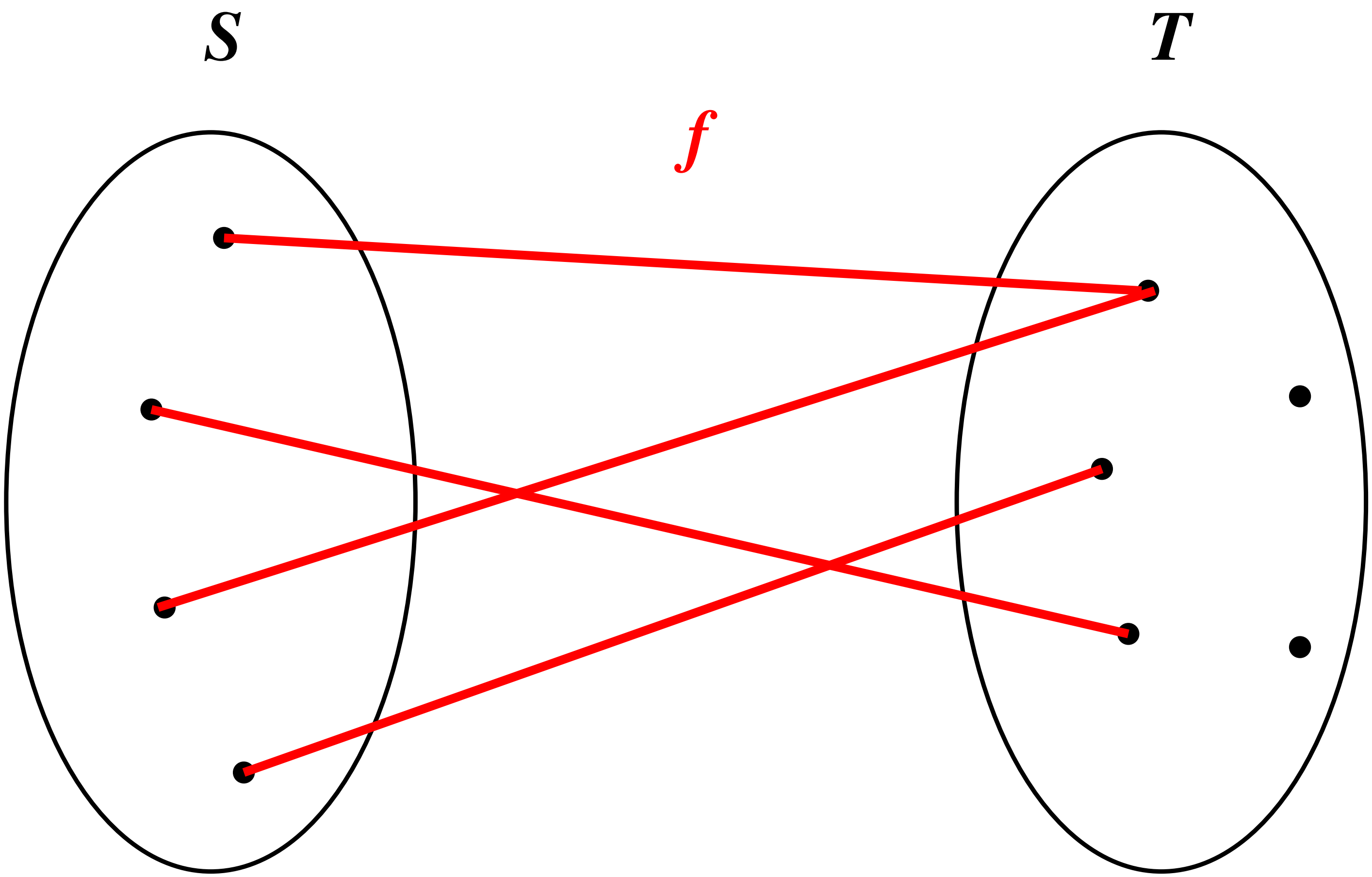
thm1 : $\mathrm{ran}(f) \neq \varnothing$

end

- We are given two sets S and T

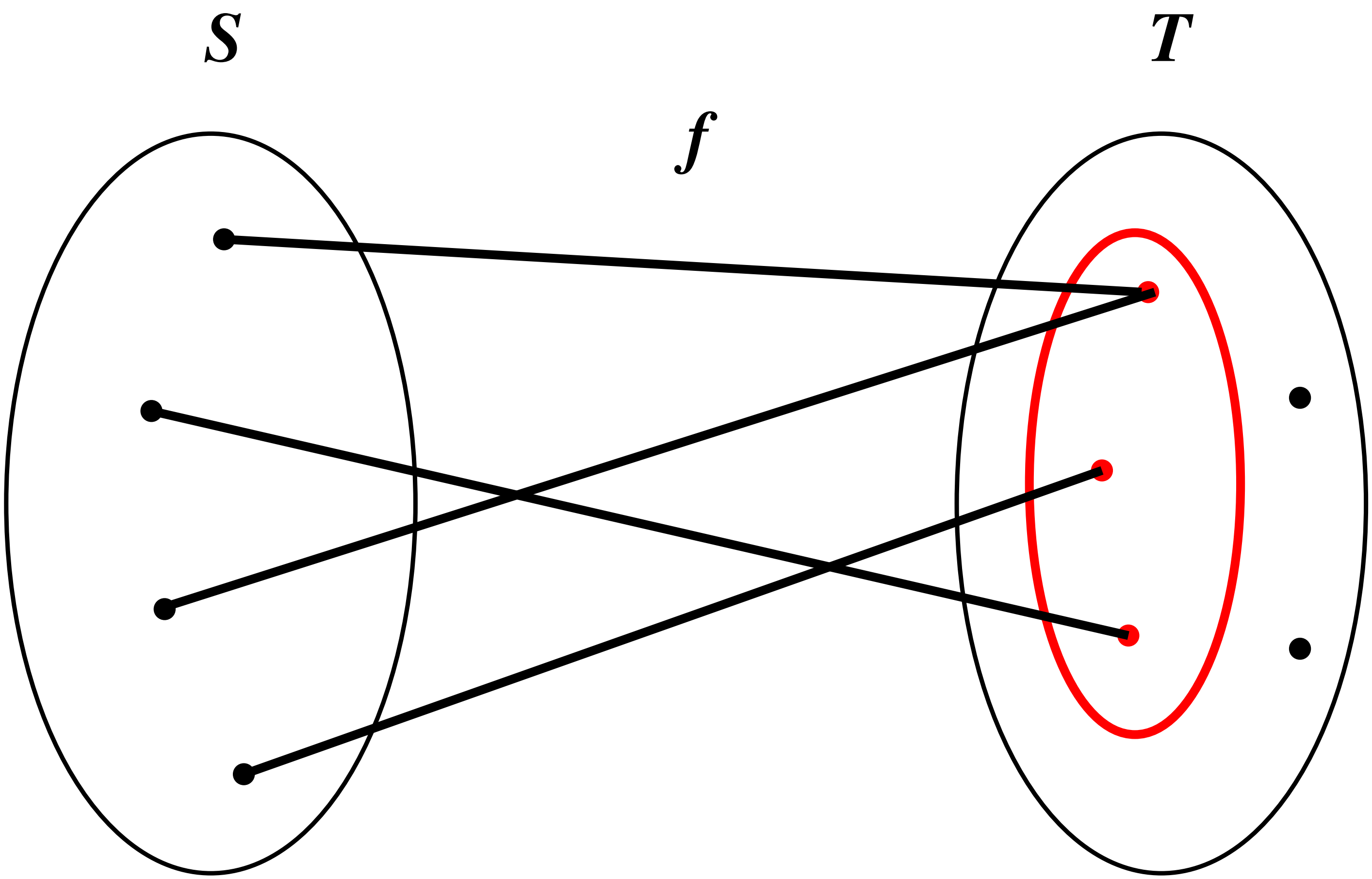


- Here is a total function f from S to T: $f \in S \rightarrow T$



Math Refresher: Total Function and Range

- Here is the range of f



D E M O (showing a context)

context
$< context\_identifier >$
sets
$< set\_identifier >$
$\ldots$
constants
$< constant\_identifier >$
$\ldots$
axioms
$< label >: < predicate >$
$\ldots$
end

Explaining Context Sections

- "**sets**" lists various sets, which define pairwise disjoint types

- "**constants**" lists the different constants introduced in the context

- "**axioms**" defines the properties of the constants

- Variable m denotes the result.

variable: m

inv0_1: $m \in \mathbb{N}$

- Next are the two events:

INIT
  begin
    $m := 0$
  end

maximum
  begin
    $m := \max(\mathrm{ran}(f))$
  end

- Event maximum presents the final intended result (in one shot)

Machine Example as presented in the tool

Machine

- variables
- invariants
- theorems
- events

machine
  maxi_0
sees
  ctx_0
variables
  i
invariants

  inv1 : $i \in 1\,..\,n$

events
  $\dots$
end

Machine (and Context) Example as presented in the tool

machine
  maxi_0
sees
  maxi_ctx_0
variables
  m
invariants

  inv1 : $m \in \mathbb{N}$

events
  ...
end

context
  maxi_ctx_0
sets
  D
constants
  n
  f
  v
axioms

  axm1 : $0 < n$

  axm2 : $f \in 1..n \rightarrow 0\,..\,M$

  thm1 : $\mathrm{ran}(f) \neq \varnothing$

end

D E M O (showing a machine)

machine
  $< machine\_identifier >$
sees
  $< context\_identifier >$
  $\ldots$
variables
  $< variable\_identifier >$
  $\ldots$
invariants
  $< label >$: $< predicate >$
  $\ldots$
events
  $\ldots$
variant
  $< variant >$
end

Explaining Machine Sections

- "**variables**" lists the state variables of the machine

- "**invariants**" states the properties of the variables

- "**events**" defines the dynamics of the transition system (next slides)

- "**variant**" is explained later

Event

- An event defines a transition of our discrete system

- An event is made of a Guard G and an Action A

- G defines the enabling conditions of the transition

- A defines a parallel assignment of the variables

Kind of Events


```
begin
  A
end
```

No guard

```
when
  G
then
  A
end
```

Simple guard

```
any x where
  G(x)
then
  A(x)
end
```

Quantified guard

begin A **end**	**when** G **then** A **end**	**any** x **where** $G(x)$ **then** $A(x)$ **end**
No guard	Simple guard	Quantified guard

Our event (so far) have no guards

```
INIT
  begin
    m := 0
  end
```

```
maximum
  begin
    m := max(ran(f))
  end
```

Summary

constants: n
f
M

axm0_1: $0 < n$

axm0_2: $f \in 1\,..\,n \rightarrow 0\,..\,M$

thm0_1: $\mathrm{ran}(f) \neq \varnothing$

variable: m

inv0_1: $m \in \mathbb{N}$

INIT
begin
$m := 0$
end

maximum
begin
$m := \max(\mathrm{ran}(f))$
end

Our Task is not Completed

- We have to perform some proofs:

  - **thm0_1** holds

  - Invariant **inv0_1** is established by event "INIT"

  - Invariant **inv0_1** is maintained by event "maximum"

  - Expression "$\mathbf{max}(\mathbf{ran}(f))$" is well-defined

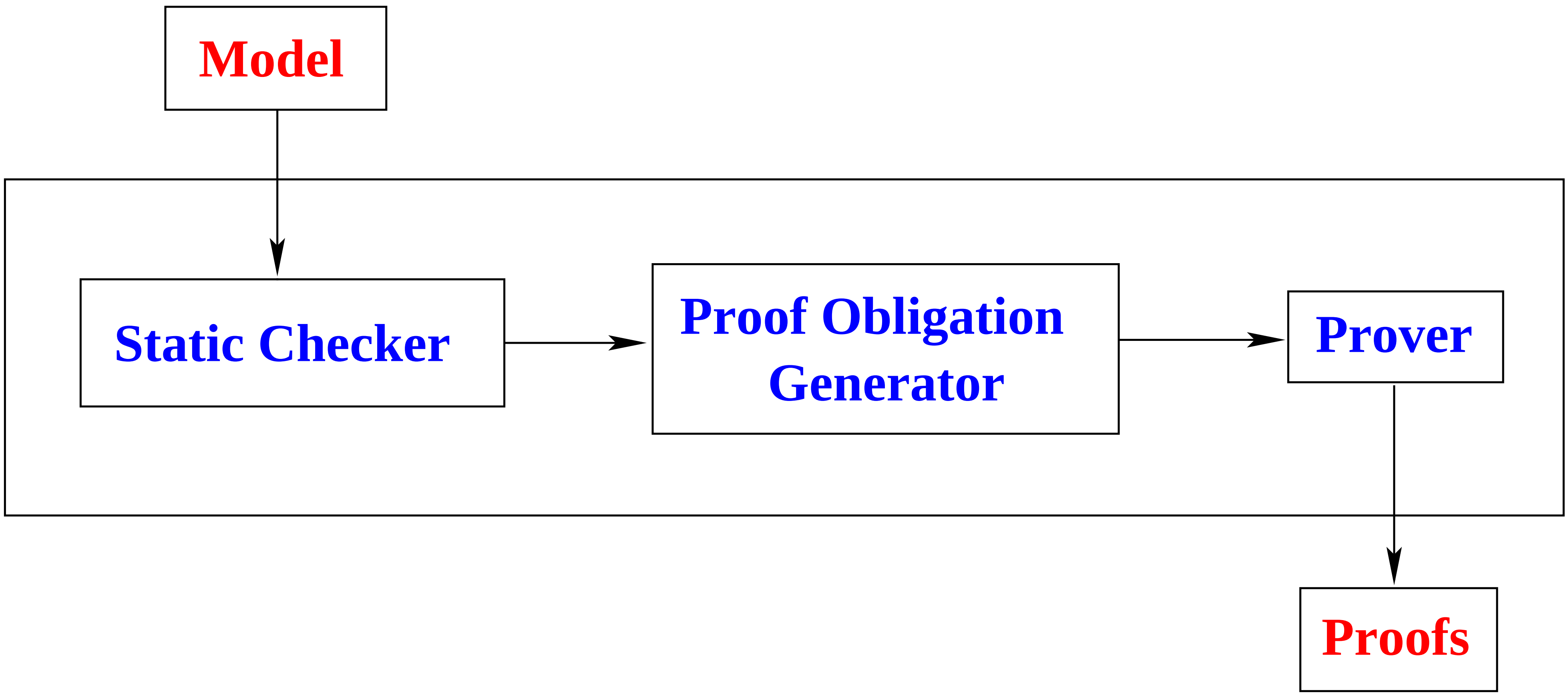
Summary of what is to be Proved

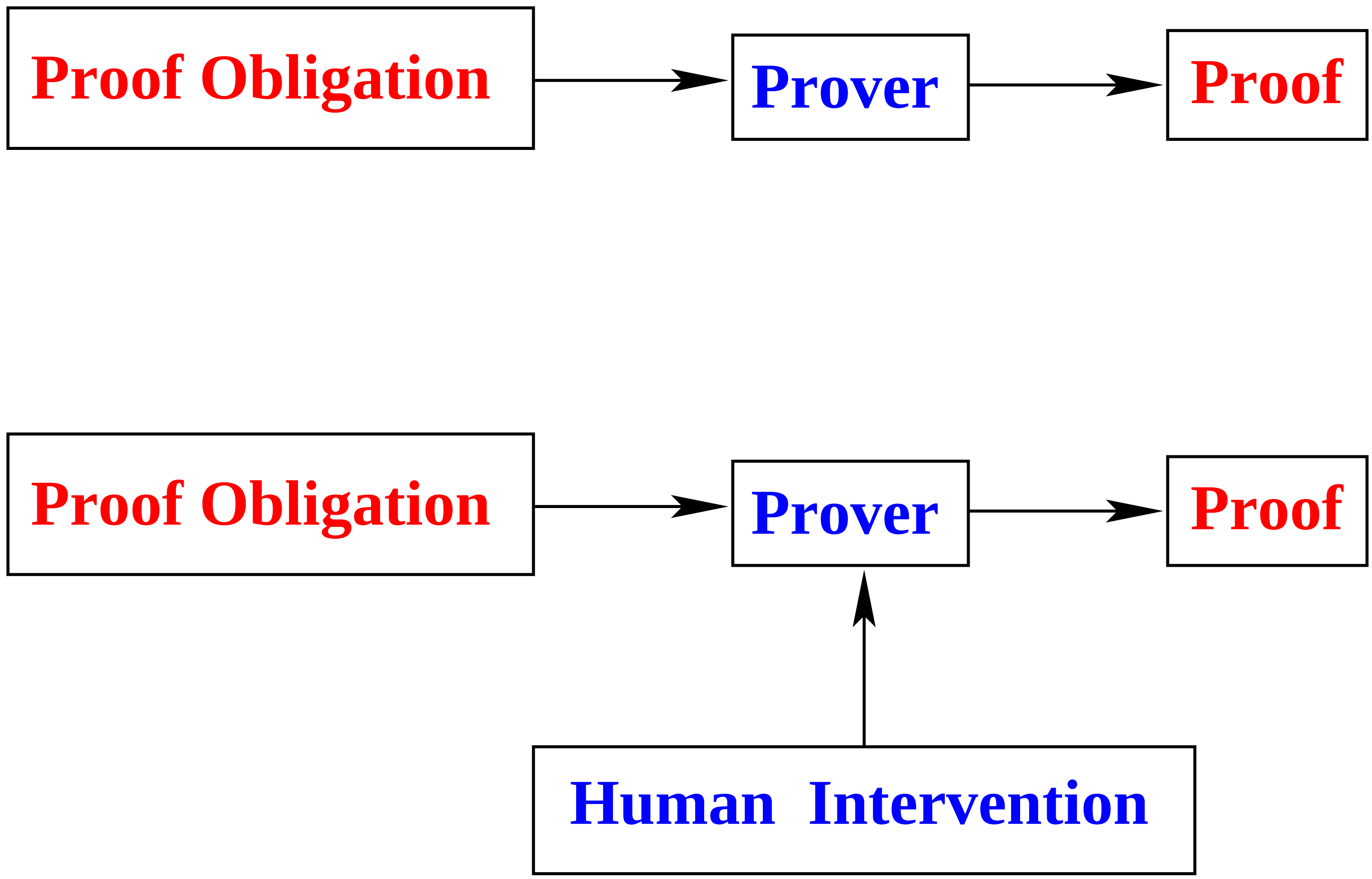
- Stated theorems

- Invariant maintenance

- Well-definedness

D E M O (showing proof obligations)


Model
Static Checker
Proof Obligation Generator
Prover
Proofs


Proof Obligation
Prover
Proof
Proof Obligation
Prover
Proof
Human Intervention

Refinement: the New Variables

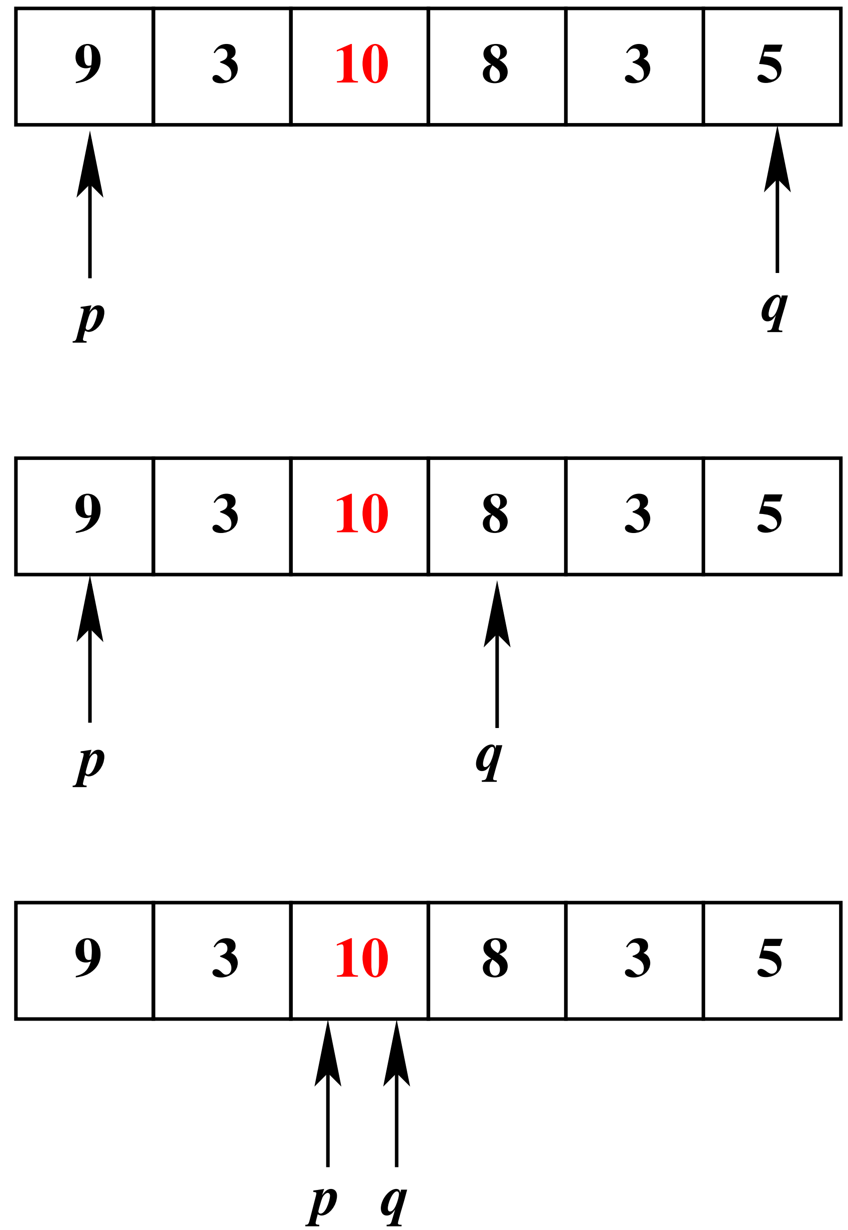
- We introduce two new variables in our model

- Variables p and q denote two indices in the domain of f.

variables: m, p, q	**inv1_1:** $p \in 1\,..\,n$ **inv1_2:** $q \in 1\,..\,n$

Initial, Intermediate, and Final Situations

The maximum is always "between" p and q



Refinement: the Main Invariant

- Interval $p \,..\, q$ is never empty (**inv1_3**)

- The maximum is always in the image of $p \,..\, q$ under f (**inv1_4**)

variables: m
p
q

inv1_1: $p \in 1 \,..\, n$

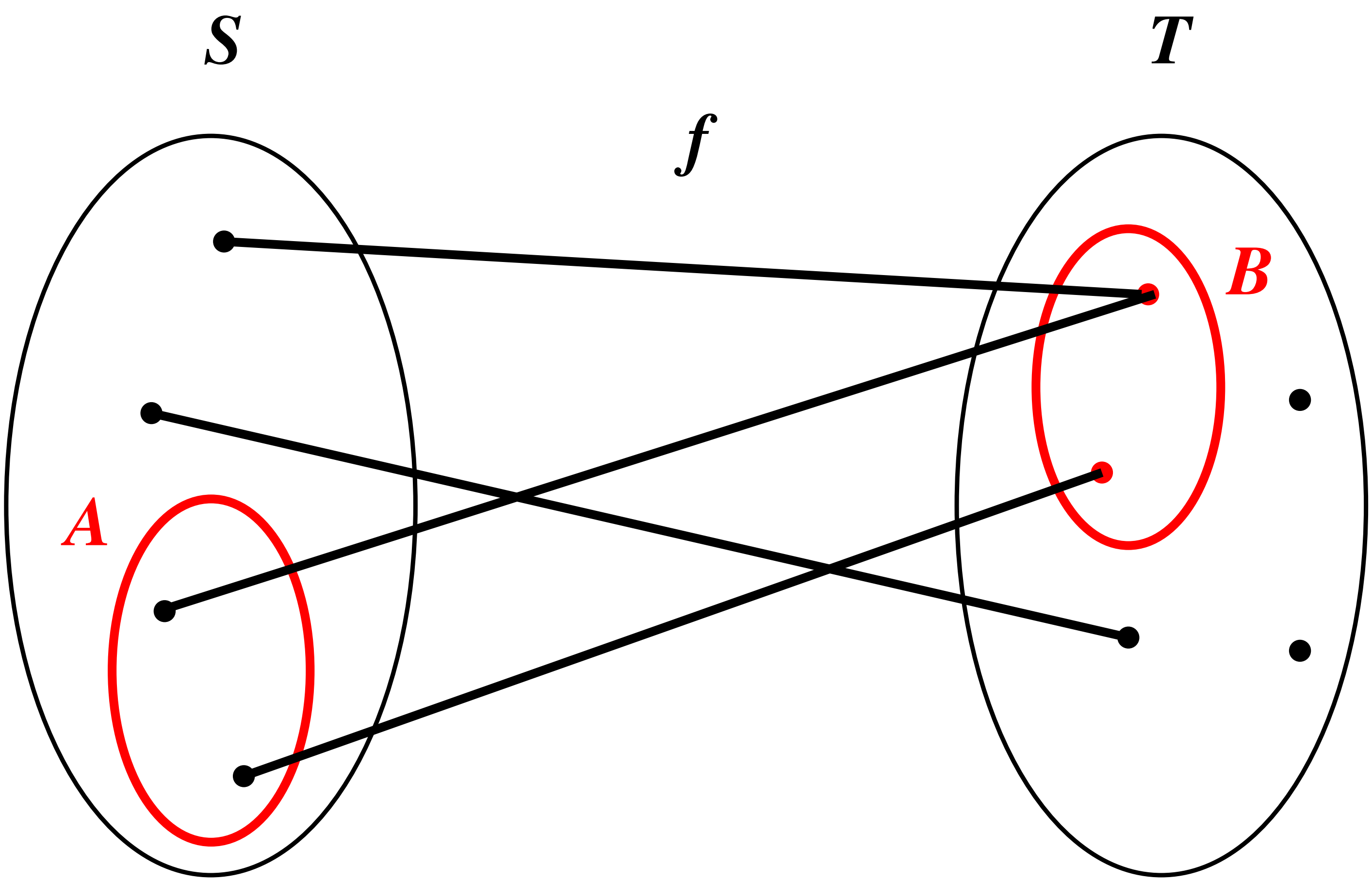
inv1_2: $q \in 1 \,..\, n$

inv1_3: $p \leq q$

inv1_4: $\max(\mathrm{ran}(f)) \in f[p \,..\, q]$

- **inv1_4** is the main invariant

- B is the image of A under f: $B = f[A]$



Refinement: Initial and Final Events

INIT	maximum
begin	**when**
$m := 0$	$p = q$
$p := 1$	**then**
$q := n$	$m := f(p)$
end	**end**

Refinement: two New Events

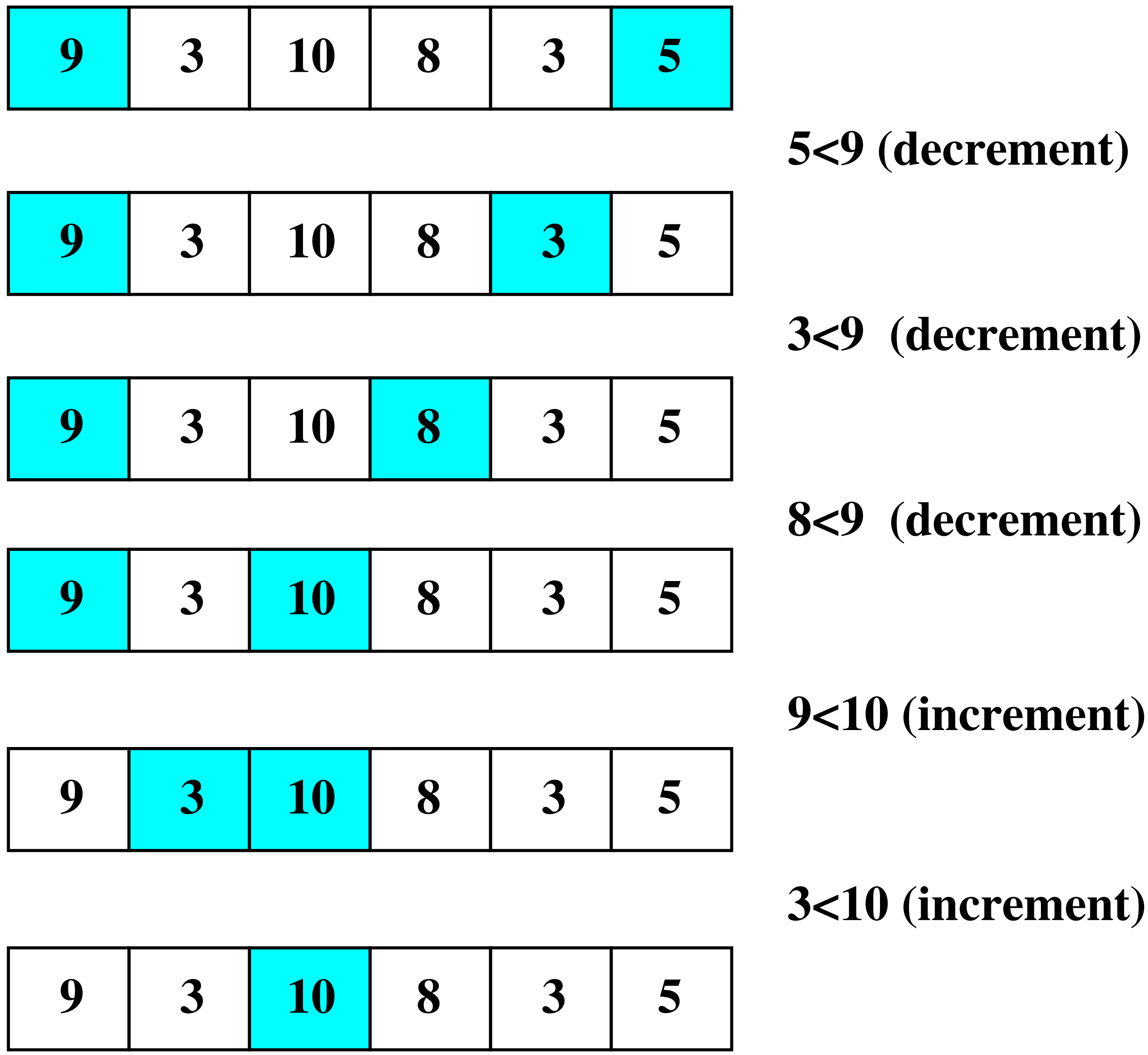
INIT
begin
$m := 0$
$p := 1$
$q := n$
end

maximum
when
$p = q$
then
$m := f(p)$
end

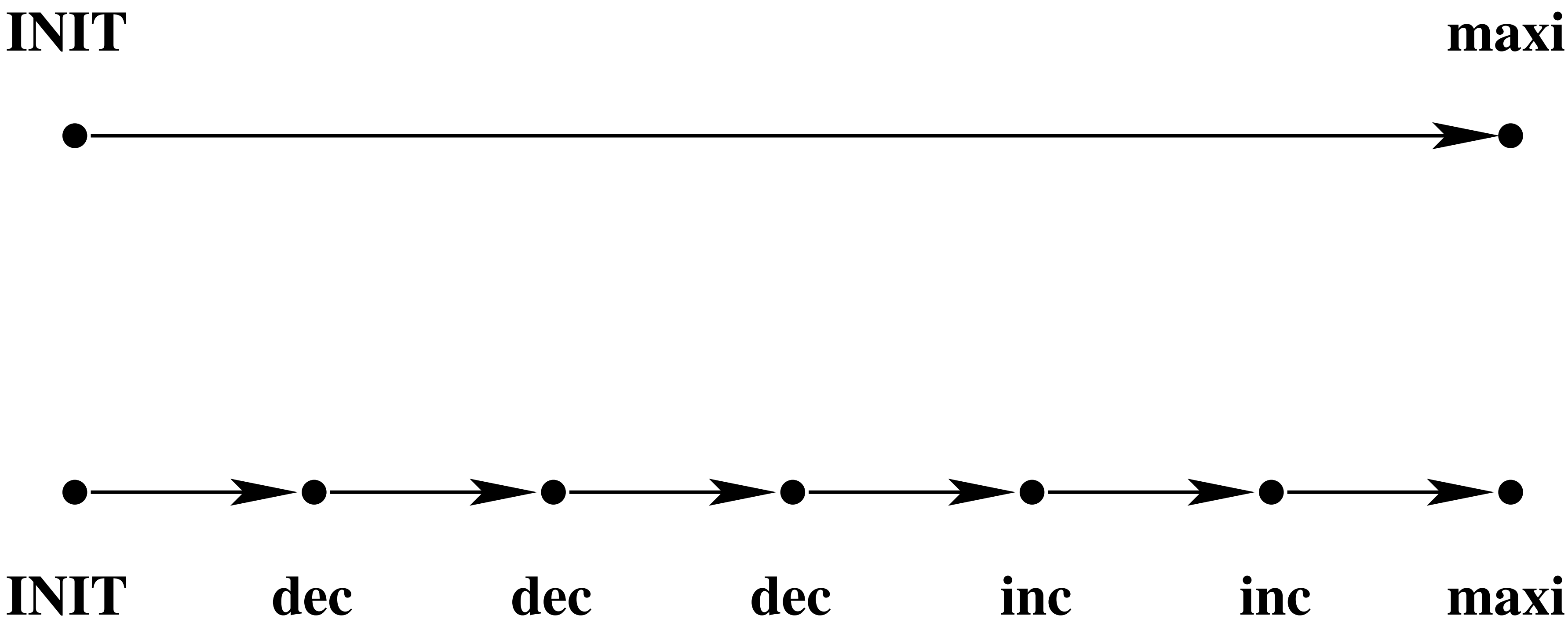
increment
when
$p < q$
$f(p) \leq f(q)$
then
$p := p + 1$
end

decrement
when
$p < q$
$f(q) < f(p)$
then
$q := q - 1$
end

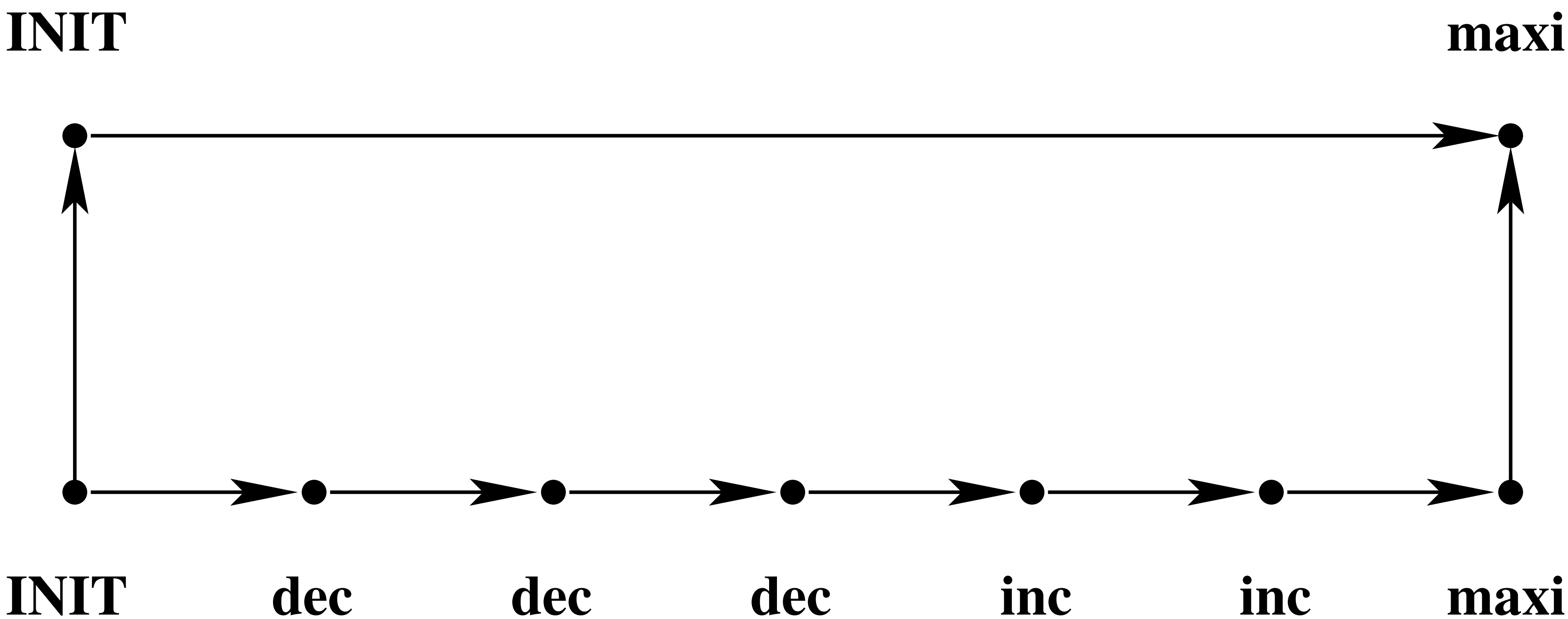
Trace


9 3 10 8 3 5
5<9 (decrement)
9 3 10 8 3 5
3<9 (decrement)
9 3 10 8 3 5
8<9 (decrement)
9 3 10 8 3 5
9<10 (increment)
9 3 10 8 3 5
3<10 (increment)
9 3 10 8 3 5

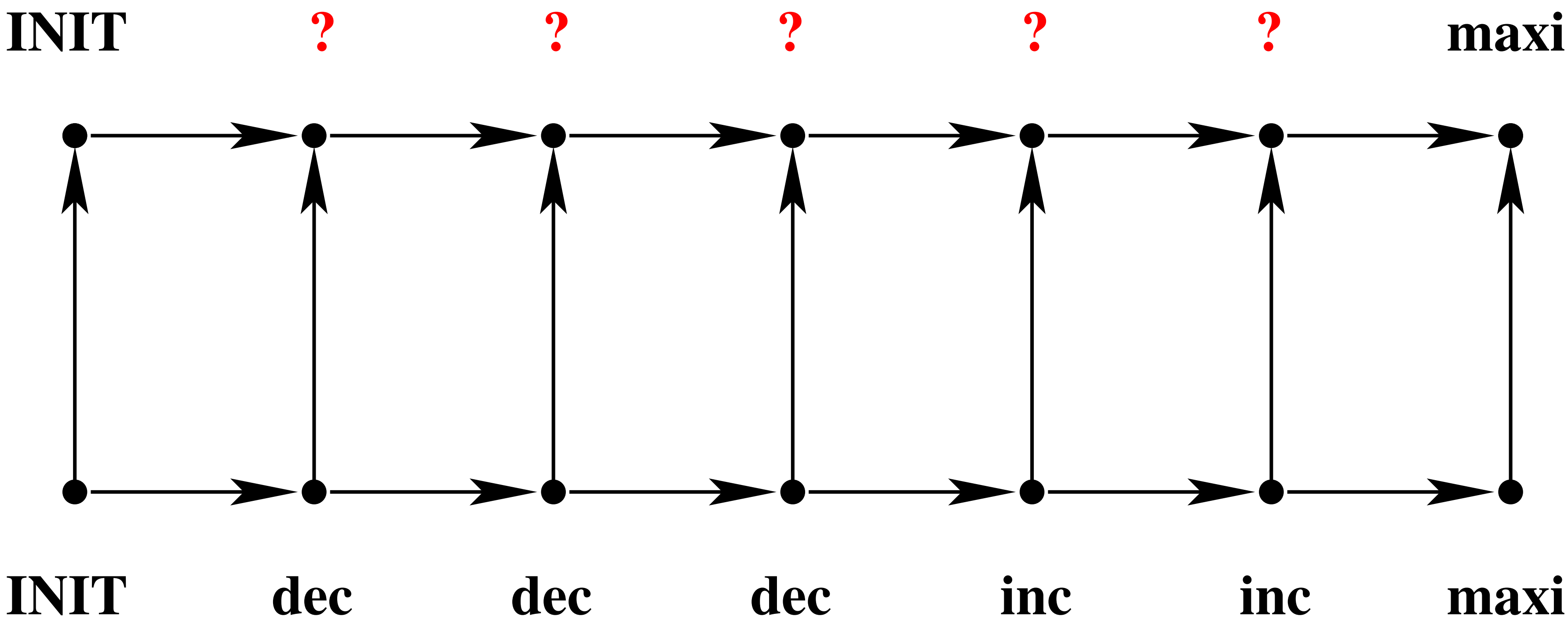
Abstract and Concrete Traces



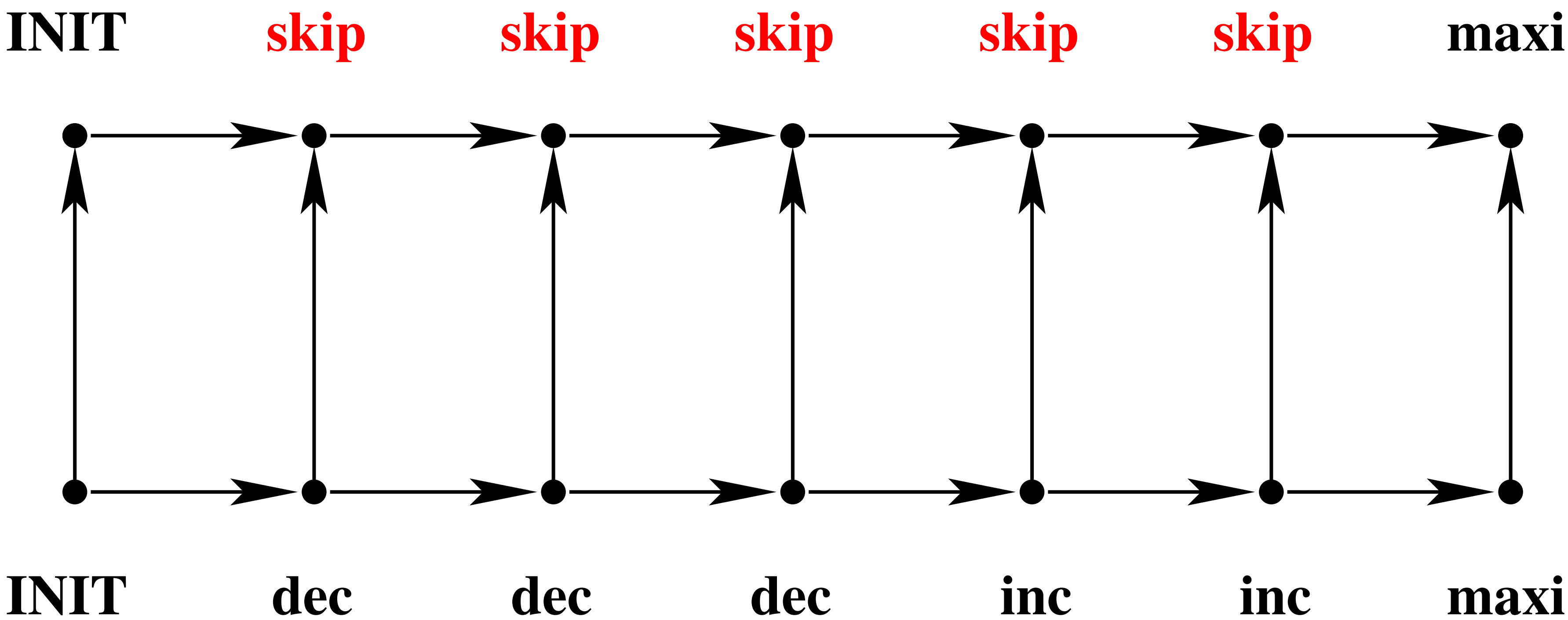
Old Events must Refine their Abstractions



How about New Events?



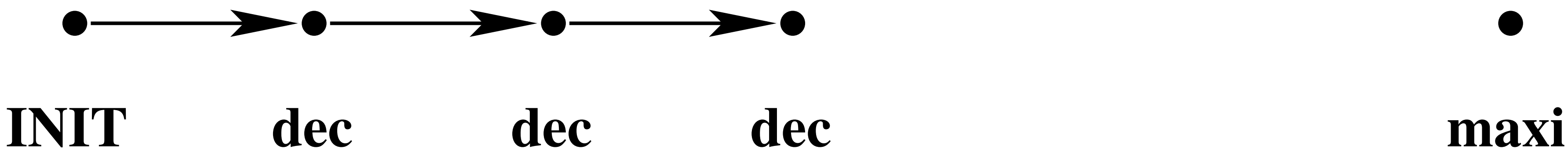
New Events refine "skip"



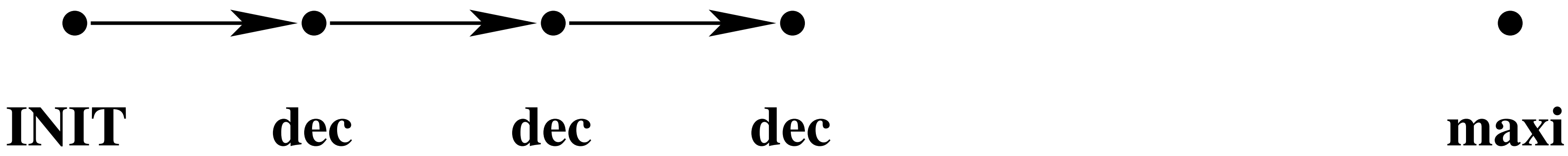
To be Proved

- Invariant maintenance
- Event refinement
    - guard strengthening
    - concrete action simulates the abstract one
- Well-definedness

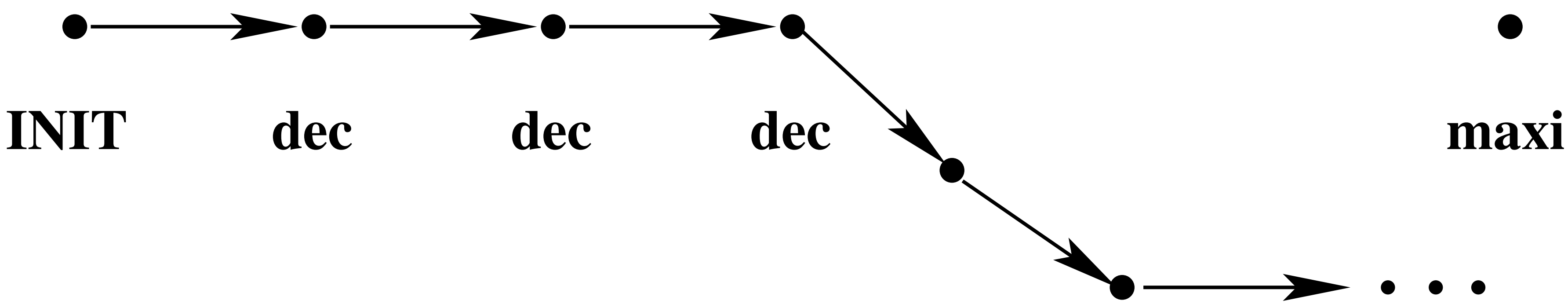
- Early deadlock



- Early deadlock



- Divergence



To be Proved (more)

- Invariant maintenance

- Event refinement

    - guard strengthening

    - concrete action simulates the abstract one

- Well-definedness

- Trace refinement

    - Disjunction of guards must hold (no early deadlock)

    - New events must be convergent (must decrease a variant)

Towards the Final Construction

INIT
begin
$m := 0$
$p := 1$
$q := n$
end

maximum
when
$p = q$
then
$m := f(p)$
end

increment
when
$p \neq q$
$f(p) \leq f(q)$
then
$p := p + 1$
end

decrement
when
$p \neq q$
$f(q) < f(p)$
then
$q := q - 1$
end

while $condition$ **do** $statement$ **end**

if $condition$ **then** $statement$ **else** $statement$ **end**

$statement \; ; statement$

$variable\_list \;\; := \;\; expression\_list$

IF Merging Rule

when
P
Q
then
S
end
	when
	P
	$\neg\, Q$
	then
	T
	end
		$\rightsquigarrow$	**when**
			P
			then
			if Q **then**
			S
			else
			T
			end
			end
				M_IF

- The two events must have been introduced at the same step

Applying Rule M_IF

```
decrement
   when
      p ≠ q
      f(q) < f(p)
   then
      q := q − 1
   end
```

```
increment
   when
      p ≠ q
      f(p) ≤ f(q)
   then
      p := p + 1
   end
```

```
decrement_increment
   when
      p ≠ q
   then
      if  f(q) < f(p)  then
         q := q − 1
      else
         p := p + 1
      end
   end
```

WHILE Merging Rule (special case)

when Q **then** S **end**	**when** $\neg\, Q$ **then** T **end**	$\leadsto$	**while** Q **do** S **end**; T	**M_WHILE**

- The first event must have been introduced at one refinement step below the second one.

Applying Rule M_WHILE (special case)

```
decrement_increment
   when
      p ≠ q
   then
      if  f(q) < f(p)  then
         q := q − 1
      else
         p := p + 1
      end
   end
```

```
maximum
   when
      p = q
   then
      m := f(p)
   end
```

```
decrement_increment_maximum
   while  p ≠ q  do
      if  f(q) < f(p)  then
         q := q − 1
      else
         p := p + 1
      end
   end;
   m := f(p)
```

WHILE Merging Rule (general case)

when P Q **then** S **end**	**when** P $\neg\, Q$ **then** T **end**	$\leadsto$	**when** P **then** **while** Q **do** S **end**; T **end**	M_WHILE

- P must be invariant under S
- The first event must have been introduced at one refinement step below the second one.

Final Rule M_INIT

- Once we have obtained an “event” without guard

- We add to it the event init by sequential composition

- We then obtain the final “program”

```
m, p, q := 0, 1, n;                          INIT
while  p < q  do
   if  f(q) < f(p)  then
      q := q − 1                             decrement
   else
      p := p + 1                             increment
   end
end;
m := f(p)                                    maximum
```

```
INIT
  begin
    m := 0
    p := 1
    q := n
  end
```

```
decrement
  when
    p < q
    f(q) < f(p)
  then
    q := q − 1
  end
```

```
increment
  when
    p < q
    f(p) ≤ f(q)
  then
    p := p + 1
  end
```

```
maximum
  when
    p = q
  then
    m := f(p)
  end
```

Exercise

- Modify the development to search for the minimum of the array

```
m, p, q := 0, 1, n;                          INIT
while p < q do
   if f(p) > f(q) then
      p := p + 1                              increment
   else
      q := q − 1                              decrement
   end
end;
m := f(p)                                     maximum
```

Conclusion (1) : Some Guidelines for Development

- Write the requirement document

- Propose a refinement strategy

- Develop the model by successive refinements and proofs

- Perform some animation (if useful)

Conclusion (2): Importance of Refinement

- Refinement allows us to build models gradually

- We build an ordered sequence of more precise models

- Each model is a refinement of the one preceding it

- A useful analogy: looking through a microscope

- Spatial (more variables) as well as temporal (more events) extensions