

Programų sistemu modeliavimas ir verifikavimas

2019 Pavasaris

Modelling and verification of software-based systems

Spring 2019

Modelling and verification of software-based systems: Course introduction

Master level course for students in Computer Science

- Course web page, lecture slides, exercises, and other materials will be available on Moodle (emokymai.vu.lt)
- Lecturer: Linas Laibinis,
e-mail: linas.laibinis@mif.vu.lt (or via Moodle)
- Office: MIF Building, Didlaukio 47, room 504

Modelling and verification of software-based systems: Course introduction

- Lectures: Tuesdays 18–20, 421 MIF-Didlaukio
- Exercise sessions/consultation/tutorials: Wednesdays 18–21, the same room
- Exercises sessions will consist of tool tutorials, solving the given modelling tasks under supervision, and presenting solutions of the (given in advance) exercises
- 5–6 units of the given exercises. The marks of the solved will give you 40% of the overall exam points

Modelling and verification of software-based systems: How to pass the course

- Final exam at the end of the course
- Exam: to solve a number of modelling tasks of various difficulty (with a pen and paper)
- The exam exercises will give you 60% of the overall exam points
- To be allowed to take the exam, at least 3 solutions should be presented/defended during the exercise sessions

Modelling and verification of software-based systems: Course goals

- Understand and apply the essential concepts behind system modelling and verification (e.g., representing a system model in some formal language, expressing the desired qualitative and quantitative system properties, etc.)
- Learn how to formalise a set of system requirements and its essential properties in the chosen modelling framework
- Learn how to formulate and verify the pre-defined system properties (functional, safety, liveness, temporal, etc.) of different types of software-based systems within the chosen framework
- Try out two different modelling and verification environments (Event-B/Rodin, Uppaal)

Sounds awfully complicated. What is all about? (again)

- It is about system models and their connection with "real things" (programs, software- or computer-based systems, the physical environment, etc.)
- It is also about model relationships with system requirements as well as (possibly) their mathematical representations
- It is also about how such models be used for analysing and verifying the systems being constructed
- Finally, it is about how existing automated frameworks can help in modelling and verifying tasks

Modelling and verification of software-based systems: Literature

- J-R. Abrial. Modelling in Event-B: System and Software Engineering. Cambridge University Press. 2010
- Rodin Tutorial. Online: <http://www3.hhu.de/stups/handbook/rodin/current/html/tutorial.html>
- Uppaal 4.0: Small Tutorial. Online: http://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf

Motivation

- Modern software-intensive systems are becoming increasingly complex
- Hard to understand and verify
- Pervasive use of computer-based systems in many critical infrastructures
- Can we rely on these systems and on the software that they are running?



System models: What are they?

- System models give us an abstracted view on the system architecture, intended behaviour, and its essential properties
- They are also a more structured and disciplined representation of the system requirements
- Modelling = methods or languages for building system models
- A variety of such methods: UML, AADL, JML, state charts, use cases, sequence diagrams, VDM, Z, B, ...

Models: What are they needed for?

- Better understanding of the domain/area/problem (the involved concepts, relationships, ...)
- "Debugging" of system requirements (finding inconsistencies, contradictions, missing information)
- "Sketching" of the overall system design (system architecture, control/data flow between components, typical solutions or patterns)
- Part of documentation (assertions, sequence diagrams, use cases, ...)
- Knowledge preservation and reuse during system upgrading, re-designing, ...
- "Blueprints" of the overall software-based system (ideally)

Other practical uses for constructed system models

- Inputs for generating test cases ([model-based testing](#))
- [Code contracts](#): runtime verification using the incorporated assertions (preconditions, class invariants, ...)
- [Static verification](#): code with the incorporated specification fragments (assertions) is statically analysed by a pre-compiler
- [Code generation](#) (in some well-defined cases)

Models of software-based systems: Summarising

- Based on using **abstraction** techniques to deal with the system complexity; Can be textual or graphical
- Intermediary between **system requirements** and its implementation
- Often reflect **patterns** of system architecture, computation, control or data flows, communication, reactions to failures
- Can be used as a basis for
 - Static or dynamic **verification** of system properties;
 - **Model-based development** (e.g., Design by contract, refinement-based approaches);
 - **Model-based system validation** (e.g., testing or simulation);
 - Documentation, preservation and transfer of **knowledge about the domain**

Formal models

- Based on formal (mathematical) semantics/representation of system objects, states and transitions
- Allow rigorous reasoning and formal verification of system properties (which is certification requirement in some critical domains)
- Many formal and semi-formal languages: Z, VDM, UML (partly), AADL (partly), B Method, Event-B, Alloy, Promela, Uppaal, etc.
- The formal semantics is based on: set theory, predicate calculus, probability theory (e.g., Markov chains and processes), timed and probabilistic automata, (labeled) state transition systems, etc.

The notion of program state

- Most of formalisms are **state-based**: the system is described as a set(type) of possible states and state transitions (changes)
- Typically, the overall system state is a collection of the current values of state variables or component attributes
- Examples of state-based formalisms: (labelled) state transition systems, timed or probabilistic automata

State transitions

- Model state transitions can be deterministic, nondeterministic or probabilistic
- Nondeterminism allows
 - abstract away the system details (the internal nondeterminism due to under-specification),
 - model the system or component environment (the external nondeterminism due to inability to control),
 - model parallel execution (using the interleaving semantics)

Formal models: examples

- An Event-B model: textual description of state-based component, its state (variables) and transitions (reactions)

Context RailwayCrossing_ctxt

POS_SET = {0, CRP, SRP, SRS, DS}, // $0 < CRP < SRP < SRS < DS$

PHASES = {Env, Train, Crossing}

BAR_POS = {Open, Closed}

Machine RailwayCrossing

Variables train_pos, phase, emrg_brakes, bar₁, bar₂

Invariants ...

Events ...

*UpdatePosition*₁ ≡

when

phase = Env \wedge train_pos < DS \wedge emrg_brakes = FALSE

then

train_pos := min({p | p ∈ POS_SET \wedge p > train_pos}) || phase := Train

end

*UpdatePosition*₂ ≡

when

phase = Env \wedge (train_pos = DS \vee emrg_brakes = TRUE)

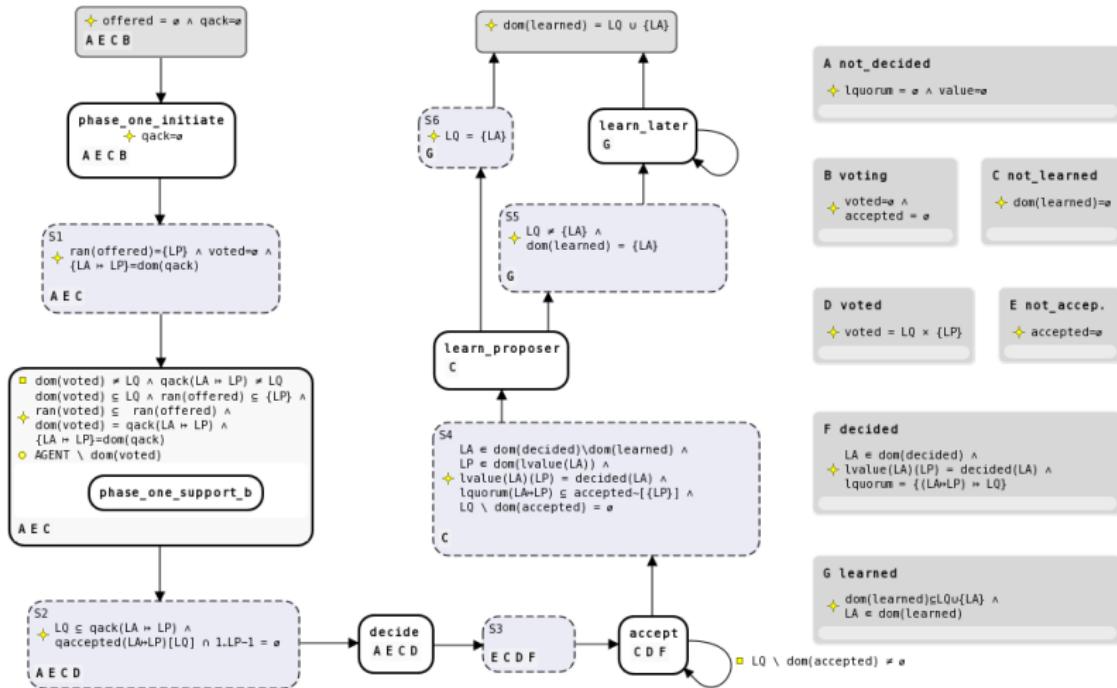
then

skip

end

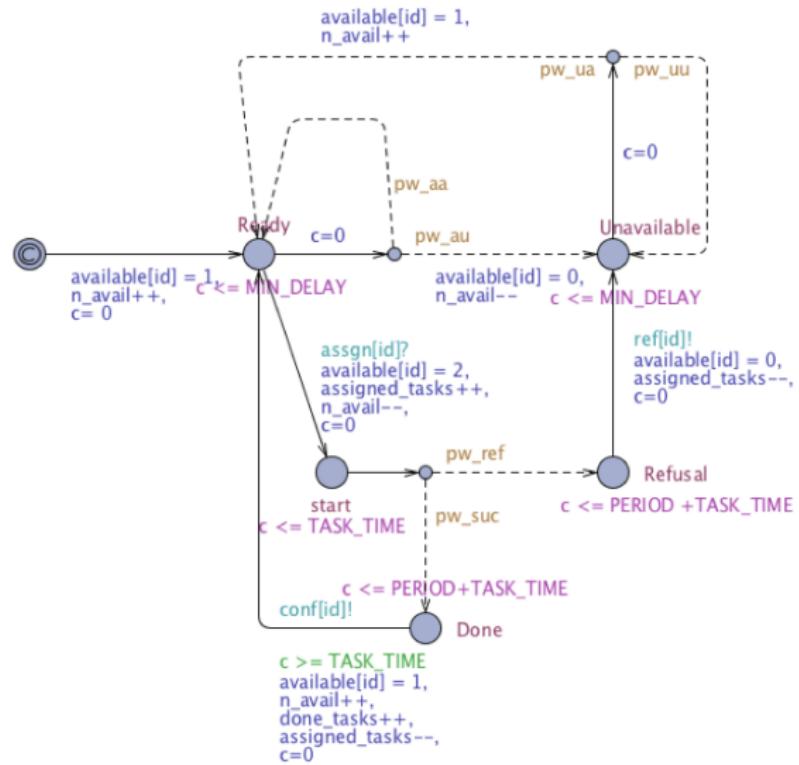
Formal models: examples

- State diagrams (states, transitions, conditions, and updates)



Formal models: examples

- Uppaal timed automata: locations (states), transitions (with conditions and updates), clock changes (within locations), probabilities on transitions



Formal models: examples

- Formal models as (incorporated) code contracts, used for static verification

```
static void Swap(int[]! a, int i, int j)
requires 0 <= i && i < a.Length;
requires 0 <= j && j < a.Length;
modifies a[i], a[j];
ensures a[i] == old(a[j]);
ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Formal verification

- Formal models are often used as **inputs** for verification of desired system properties
- System properties (usually **formalised system requirements**) are typically incorporated as parts of system models
- Such extended models are **checked or mathematically proved** to adhere/conform to the given properties
- Since formal verification can be quite a complex task, **automated tools and environments** are used to facilitate such verification

Formal verification (cont.)

- Provides automated assurance relying on theorem proving and model checking
- Theorem proving: mathematically proving the model properties for all system transitions; Requires bigger efforts, yet stronger assurance
- Model checking: mechanically checking all the system states based on the reachability graph; Quicker results, yet likely state explosion
- Many automated tools: Atelier-B, Rodin platform, Spin, Uppaal, PRISM, nuSMV, Perfect Developer, AADL, Mobius, etc.

Verification vs validation

- Validation provides **arguments/evidence** (not proof!) that the system behaves as expected
- Not as strong assurance as verification
- Examples of validations: testing, statistical model checking, runtime simulation, safety/security cases (as secondary structuring of the evidence)
- **Statistical model checking:** not all states are checked; the result is returned with some confidence/probability of its correctness
- **Runtime simulation:** a system prototype (with probabilistic / quantitative mathematical estimates) is run many times and statistics is accumulated

Different types of software-based systems

- Different domains, characteristics, architectures, criticality levels
- Closely tied up with the **physical world**: control systems, monitoring systems, embedded systems, cyber-physical systems
- Various **component architectures**: layered architectures, service-based architectures, cloud-based systems, multi-agent systems, adaptive and mode-rich systems
- **Communicating systems**: (tele)communication networks, satellite systems
- **Dependable systems**: safety-critical systems, fault-tolerant systems, resilient (adaptable) systems
- Different systems ⇒ different requirements, different desired properties

Types of system properties

- **Functional correctness**: preservation of pre- and post-conditions, system invariants
- **System progress** or liveness: temporal (reachability) properties
- **Timing** properties (reachability + time bound)
- **Quantitative (probabilistic) assessment** of numerical system characteristics such as failures, performance, service time
- Based on the previous results, comparison of different system architectures or configurations

The formalisms covered in this course

- Two formalisms: [Event-B](#) (mostly) and [Uppaal](#) (the last few lectures)
- Event-B is based on model refinement and verification by theorem proving. Focuses on reactive (event-based) systems and functional correctness properties
- Uppaal is based on timed automata and verification by model checking. Focuses on real-time systems and timing/temporal properties

Why Event-B is chosen for this course?

- Easy to learn
- Models cover a wide class of software-systems (reactive, distributed, asynchronous)
- Good and freely available industrial-strength tool support, developed by non-profit companies
- Support of gradual model refinement (easier to handle complexity and gradually introduce implementation decisions)
- Many available tool extensions (bridging with different notations, formalisms and provers/solvers)

Event-B: Historical note

- The B Method: invented in 1990-s by J.-R. Abrial to formally specify and develop sequential systems correct by construction
- from 2000: Event-B – extension of the B Method for reactive and asynchronous systems
- from 2007: The Rodin Platform – free industrial-strength tool support for Event-B
- Successful use of Event-B in the railway domain (e.g., verification of several driver-less metro lines in Paris)

- Event-B has been successfully used in development of several complex real-life applications;
- It adopts **top-down development** paradigm based on refinement;
- **Refinement** process: we start from an abstract formal specification and gradually transform it into its implementation by a number of correctness-preserving steps
 - It allows to structure complex requirements;
 - Small transformations simplify verification;
 - Verification by theorem proving does not lead to state explosion.

- The dynamic system behaviour is described in terms of **guarded commands (events)**:
 - Stimulus → response.
- General form of an event:

WHERE *guard* THEN *action* END

where

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

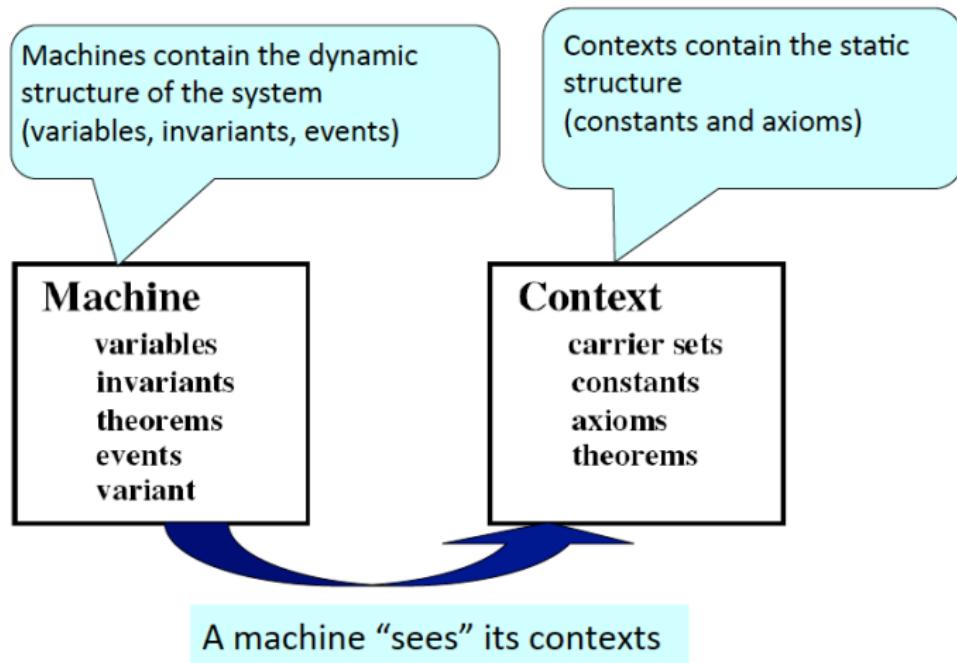
Modelling in Event-B

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

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

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

A system model in Event-B



- The Rodin platform provides **automated support** for modelling and verification in Event-B
- Automated checks for **model consistency and correct syntax**
- Automatic **generation** of the necessary **proof obligations** (verification conditions)
- Verification is based on available **automatic and interactive provers**
- A number of **extensions**, bridging with different notations and provers

Installation and quick starting of the Rodin platform

- Go to <https://sourceforge.net/projects/rodin-b-sharp/>
- Download the latest stable version (3.4) for your OS. Usually automatically suggested on the top
- Unarchive the downloaded file and move the created folder to the place you like
- Start the executable file. When asked, choose (preferably create a new) folder for your Event-B projects
- Go to the Install New Software in the Help menu, choose the Atelier B provers site and install the Atelier B provers
- Restart the Rodin platform
- Read the Rodin tutorial document from
<http://www3.hhu.de/stups/handbook/rodin/current/html/tutorial.html>

Lecture 2: Outline

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

System Modelling vs System Design

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

Engineering, not blacksmithing

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

Event-B

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

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

In turn, the model events consist of:

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

Events and Requirements

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

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

Formal development by refinement

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

- Abstract modelling
 - Helps to cope with complexity;
 - Focus on stating requirements and assumptions;
 - Allows us to spot requirements ambiguities and contradictions.
- Refinement
 - Elaboration on abstract models;
 - Structuring of requirements;
 - (Automated) proof of adherence to the abstract model.

Modelling in Event-B

- The dynamic system behaviour is described in terms of **guarded commands (events)**:
 - Stimulus → response.
- General form of an event:

WHERE *guard* THEN *action* END

where

- *guard* is a state predicate defining when an event is enabled;
- *action* is (possibly non-deterministic) update of state variables.
- If the event guard is missing (i.e., always true), an event can be simply defined as:

BEGIN *action* END

Modelling operations in Event-B (cont.)

- General form of a parameterised event:

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

where

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

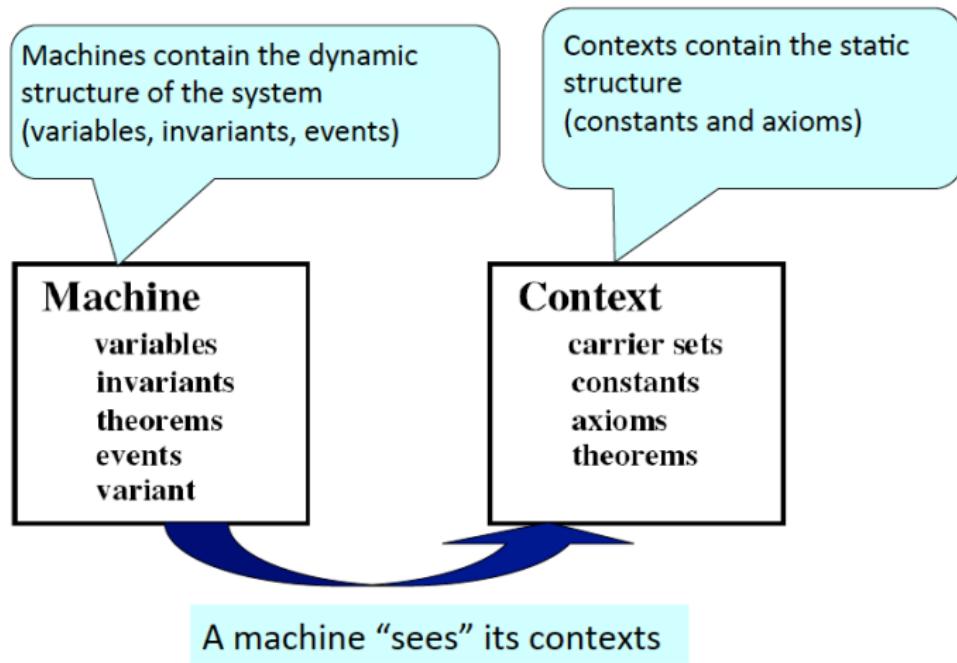
Modelling in Event-B

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

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

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

A system model in Event-B



Structure of a model context component

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

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

Structure of a model machine component

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

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

A trivial example: Booking system

Simple requirements

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

A trivial example: Booking system

MACHINE

Booking

VARIABLES

seats

INVARIANT

seats $\in \mathbb{N} \wedge \text{seats} \leq 1000$

INITIALISATION

seats := 1000

EVENTS

book =

WHEN *seats* > 0 **THEN** *seats* := *seats* - 1 **END**

cancel =

BEGIN *seats* := *seats* + 1 **END**

END

A bit less trivial (and consistent) Booking system

CONTEXT

`Booking_context`

CONSTANTS

\max_seats

AXIOMS

$\max_seats \in \mathbb{N}$

$\max_seats > 0$

END

A bit less trivial (and consistent) Booking system

MACHINE

Booking

SEES

Booking_context

VARIABLES

seats

INVARIANT

$seats \in \mathbb{N} \wedge seats \leq max_seats$

INITIALISATION

$seats := max_seats$

EVENTS

book =

WHEN $seats > 0$ THEN $seats := seats - 1$ END

cancel =

WHEN $seats < max_seats$ THEN $seats := seats + 1$ END

END



Yet another improved Booking system

CONTEXT

`Booking_context`

CONSTANTS

$\max_seats, \max_tickets$

AXIOMS

$\max_seats \in \mathbb{N}$

$\max_seats > 0$

$\max_tickets \in \mathbb{N}^1$

$\max_tickets = 5$

END

Yet another improved Booking system

The header stays the same.

MACHINE

Booking

VARIABLES

seats

INVARIANT

$seats \in \mathbb{N} \wedge seats \leq max_seats$

INITIALISATION

$seats := max_seats$

...

Yet another improved Booking system (cont.)

The events become parameterised.

...

EVENTS

book = ANY n

WHERE $n > 0 \wedge n \leq \text{max_tickets} \wedge n \leq \text{seats}$

THEN $\text{seats} := \text{seats} - n$ END

cancel = ANY n

WHERE $n > 0 \wedge \text{seats} + n \leq \text{max_seats}$

THEN $\text{seats} := \text{seats} + n$ END

END

Event-B mathematical basis: predicate calculus

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

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

Mathematical basis: predicate logic

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

Event-B mathematical basis: set theory

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

Some set constants and operations

Graphical notation, followed by the equivalent ascii notation:

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

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

Some other set expressions

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

A simple example: Request server

Some requirements of the system:

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

A simple example: Request server (cont.)

CONTEXT

Server_context

SETS

REQUESTS

MACHINE Server

SEES Server_context

VARIABLES

received, handled, completed

INVARIANT

$received \subseteq REQUEST \wedge handled \subseteq REQUEST \wedge$
 $completed \subseteq REQUEST \wedge completed \subseteq handled \wedge$
 $handled \subseteq received$

INITIALISATION

$received, handled, completed := \emptyset, \emptyset, \emptyset$

...



Abstract sets

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

Example: Request server (cont.)

EVENTS

receive = ANY *rr*

WHERE *rr* ∈ REQUEST

THEN *received* := *received* ∪ {*rr*} END

handle = ANY *rr*

WHERE *rr* ∈ *received*

THEN *handled* := *handled* ∪ {*rr*} END

reject = ANY *rr*

WHERE *rr* ∈ *received* ∧ *rr* ∉ *handled*

THEN *received* := *received* \ {*rr*} END

acknowledge = ANY *rr*

WHERE *rr* ∈ *handled*

THEN *completed* := *completed* ∪ {*rr*} END

END

An example for the exercise session: a vending machine

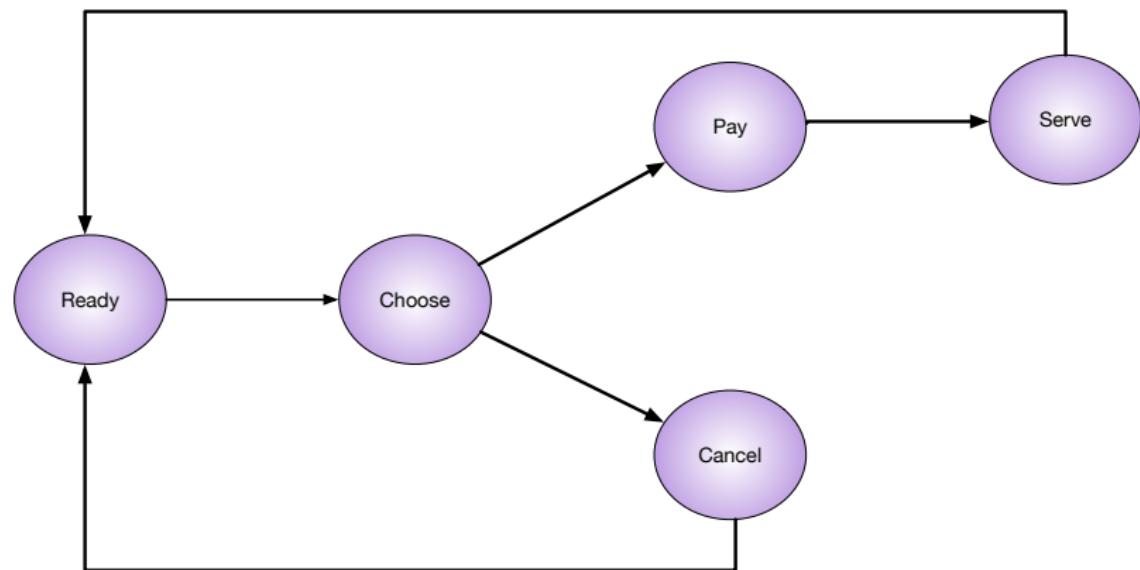
- ① The vending machine serves the customer a product from a number of available choices;
- ② The customer may select one of the available product choices;
- ③ After selection, the customer may proceed by paying for the selected product or cancelling the selection;
- ④ After sufficient payment, the selected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

Example: a vending machine (cont.)

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

Example: a vending machine (cont.)

An expected system usecase:



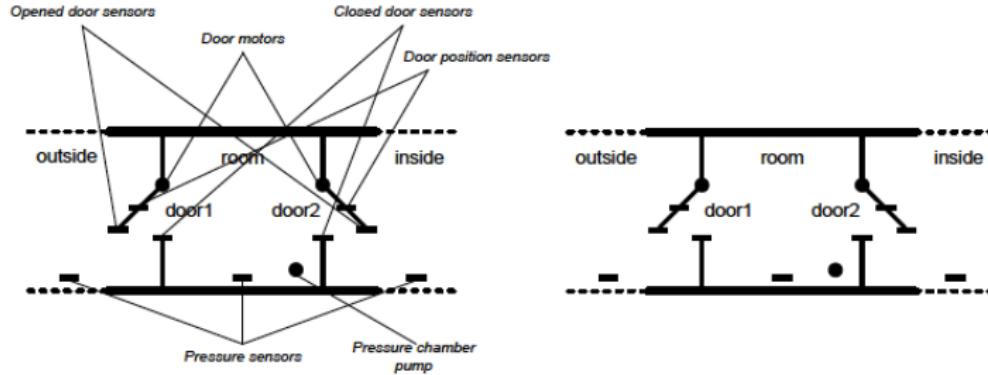
Lecture 3: Outline

- Control systems
- Case study: the sluice system
- Model refinement: simple case
- Relations (introduction)
- Examples

Control systems

- We will look into different types of computer-based systems: how we can model them, express and verify their essential properties
- Control systems: the systems that manage, regulate and command the behaviour of some physical devices or processes
- Typically, they are based on feedback loop, monitoring the state of physical devices or processes (via sensors) and reacting, if necessary, to the state changes by making appropriate commands/signals (to the controlled devices – actuators)
- Control systems often belong to the class of safety-critical systems
- Thus, safety properties (safety invariants) are essential and should be verified

Case study: sluice gate control system



- Sluice connects areas with dramatically different pressures;
- It is unsafe to open a door unless the pressure is levelled between the connected areas;
- The purpose of the system is to operate doors safely by adjusting the pressure in the room.

Example: sluice gate system requirements

- ① The purpose of the system is to allow a user to safely travel between inside or outside areas;
- ② The system has three locations - outside, middle and inside;
- ③ The system has two doors - door 1, connecting the outside and middle areas, and door 2, connecting the middle and inside areas;
- ④ A pump is located in the middle area;
- ⑤ Pressure in the inside area is always PRESSURE_LOW;
- ⑥ Pressure in the outside area is always PRESSURE_HIGH;

Example: sluice gate system requirements (cont.)

- ⑦ The middle area has a pressure sensor reporting the current pressure;
- ⑧ Both doors are equipped with sensors reporting the status of a door;
- ⑨ There are two types of sensors for each door : switch-type (binary) and value (in the range 0..100) sensors to indicate the door position;
- ⑩ The pump changes the pressure in the middle area;
- ⑪ When the pump is set to the mode PUMP_IN, it slowly increases the pressure in the middle area;
- ⑫ When the pump is set to the mode PUMP_OUT, it slowly decreases the pressure in the middle area;

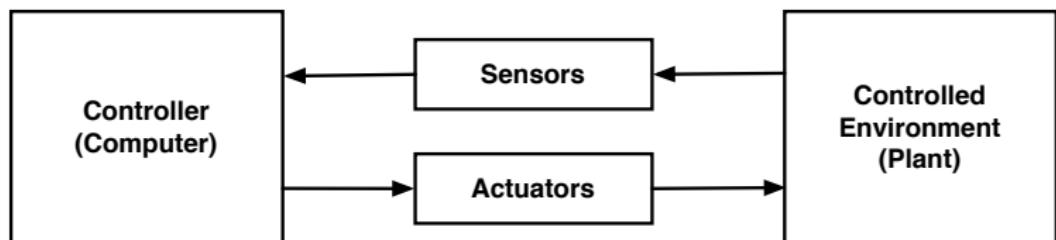
Example: sluice gate system requirements (cont.)

- ⑬ When in the mode PUMP_IN mode, the pump automatically stops when the pressure reaches PRESSURE_HIGH;
- ⑭ When in the mode PUMP_OUT mode, the pump automatically stops when the pressure reaches PRESSURE_LOW;
- ⑮ At most one door is open at any moment;
- ⑯ The outside door (door 1) can be opened only when the middle area pressure is PRESSURE_HIGH;
- ⑰ The inside door (door 2) can be opened only when the middle area pressure is PRESSURE_LOW;
- ⑱ Pressure may only be changed when both doors are closed.

Sluice example: a control system

The sluice system is an instance of a control system.

The general structure of control systems:

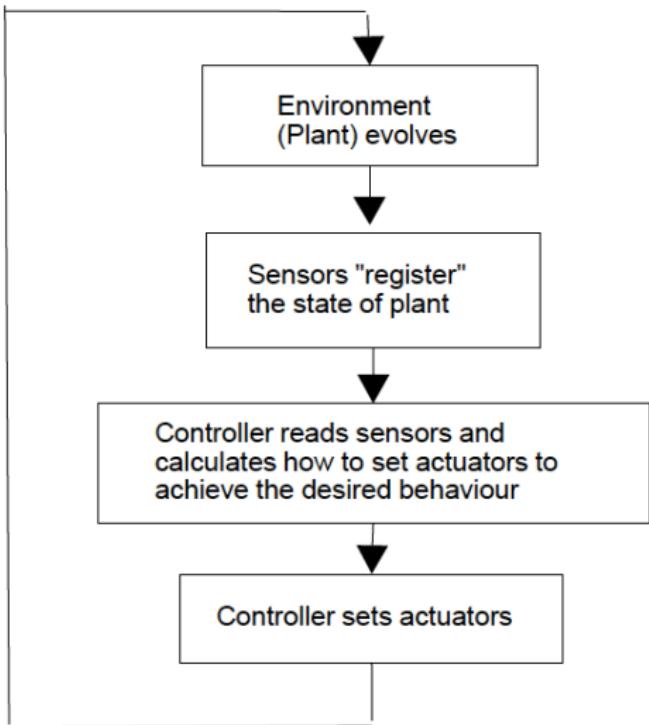


Modelling Control Systems

The control systems are cyclic:

- get inputs from the sensors,
- process them;
- output new values to the actuators.

The overall behaviour of the system is an alternation between the events modelling plant evolution and controller reaction.

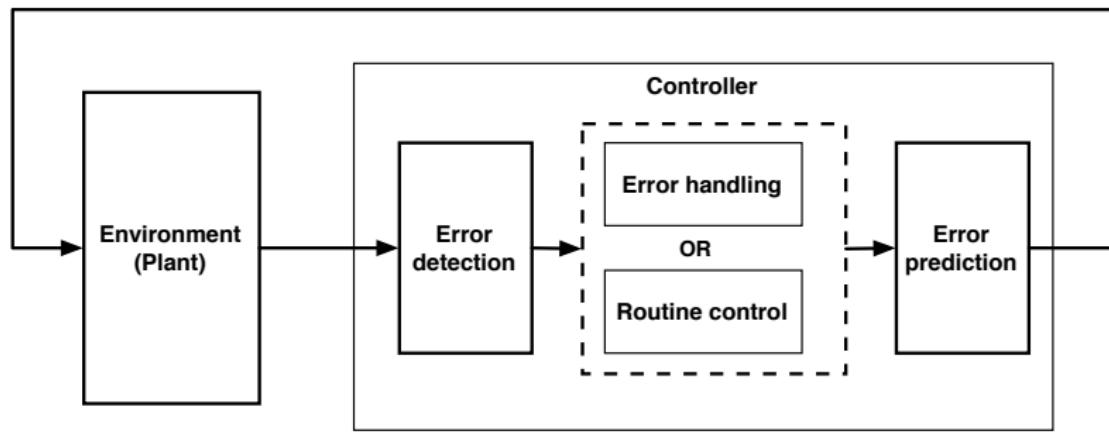


- Safety cannot be achieved without fault tolerance (FT);
- FT often relies on redundancy of sensors or actuators;
- Main goal of FT: prevent propagation of a fault to system boundaries (and potentially jeopardise safety);
- Steps of fault tolerance: error detection and error recovery;
- General principle of error detection: find a discrepancy between the expected state of a fault-free system and the observed state.

Modelling a controller

To ensure fault tolerance, its cyclic execution is often split into three steps:

- error detection based on read sensor values;
- update of its internal state and decision on possible control actions (based on both the sensor values and error detection results);
- prediction of the expected sensor values for the next cycle.



Modelling and verifying system dependability

- Ensuring dependability of complex control systems is challenging;
- Formal modelling and refinement in Event-B helps to structure complex requirements and develop systems that are correct and safe by construction;
- How to capture the system requirements in a formal model?
- Too much complexity to model everything at once \Rightarrow complexity can be handled by developing the system model at different abstraction levels, starting with the simplest one and then refining it

An abstract model of the sluice system

CONTEXT

c0

SETS

DOOR

PRESSURE

CONSTANTS

Open

Closed

Low

High

AXIOMS

partition(DOOR, {Open}, {Closed})

partition(PRESSURE, {Low}, {High})

END

- *partition* is a shorthand definition of a set that can be partitioned into separate, disjoint parts (subsets)
- If the parts (subsets) are singleton sets, e.g., of the form $\{element\}$, such a definition defines of an enumerated set
- For instance, $partition(Set, \{element1\}, \{element2\}, \{element3\})$ is a shorthand for the axioms
 - $element1 \in Set$,
 - $element2 \in Set$,
 - $element3 \in Set$,
 - $element1 \neq element2$,
 - $element1 \neq element3$,
 - $element2 \neq element3$,
 - $Set = \{element1, element2, element3\}$.
- In general, any disjoint subsets instead of $\{element\}$, e.g., $partition(ITEM, Available, Sold)$

An abstract model of the sluice system (cont.)

MACHINE

$m0$

SEES

$c0$

VARIABLES

$door1, door2, pressure$

INVARIANT

$door1 \in DOOR \wedge door2 \in DOOR$

$pressure \in PRESSURE$

$\neg(door1 = Open \wedge door2 = Open)$

$door1 = Open \Rightarrow pressure = High$

$door2 = Open \Rightarrow pressure = Low$

INITIALISATION

$door1, door2 := Closed, Closed$

$pressure := PRESSURE$

Event-B conventions (cont.)

- $x, y, \dots := Exp_1, Exp_2, \dots$ stands for a multiple parallel assignment
- The variables x, y, \dots should be separate (distinct)
- $x : \in Set$ is a simple case of a non-deterministic assignment, when any value from the given set can be assigned to the variable
- It is very useful, when we do not know, cannot control, or do not care, which exact resulting value will be used to update the variable

An abstract model of the sluice system (cont.)

EVENTS

open1 =

WHEN *door1* = *Closed* \wedge *door2* = *Closed* \wedge *pressure* = *High*
THEN *door1* := *Open* **END**

close1 =

WHEN *door1* = *Open* **THEN** *door1* := *Closed* **END**

open2 =

WHEN *door1* = *Closed* \wedge *door2* = *Closed* \wedge *pressure* = *Low*
THEN *door2* := *Open* **END**

close2 =

WHEN *door2* = *Open* **THEN** *door2* := *Closed* **END**

...

An abstract model of the sluice system (cont.)

...

pressure_low =

WHEN *door1* = *Closed* \wedge *door2* = *Closed*

THEN *pressure* := *Low* **END**

pressure_high =

WHEN *door1* = *Closed* \wedge *door2* = *Closed*

THEN *pressure* := *High* **END**

END

Refinement in Event-B

- A way to gradually develop formal models, elaborating on missing implementation details
- Allows to model the system at different abstraction levels, handle its complexity, and structure its requirements
- Consistency of model refinements is supported by the Rodin platform

Refinement in Event-B (cont.)

- Defined separately for a context and a machine;
- For a context component, it is called *extension*;
- Context extension allows
 - introducing new data structures (sets and constants), as well as
 - adding more constraints (axioms) for already defined ones.

Refinement in Event-B (cont.)

- For a machine component, there are several possible kinds of refinement:
 - simple extension of an abstract model by new variables and events (*superposition refinement*);
 - constraining the behaviour of an abstract model (*refinement by reducing model non-determinism*);
 - replacing some abstract variables by their concrete counterparts (*data refinement*);
 - a mixture of those.

Superposition refinement

Probably the simplest way to refine a model by

- Adding new variables and events;
- Reading and updating new variables in old event guards and actions;
- Interrelating new and old variables by new invariants.

Restriction: the old variables cannot be updated in new events!

Sluice example: the first refinement

The goal is to abstractly model the feedback loop of a control system. An example of superposition refinement:

- Introduce a new type $PHASE = \{Env, Det, Cont, Pred\}$;
- Add new variables $phase \in PHASE$ and $failure \in BOOL$;
- Introduce new events *Environment*, *Detection* and *Prediction* with the corresponding guards;
- Strengthen guards of the old events, making the events a part of the controller phase;
- Introduce new events *stop* and *other_control*.

The sluice system: a refined model

CONTEXT

c1

EXTENDS

c0

SETS

PHASE

CONSTANTS

Env

Det

Cont

Pred

AXIOMS

$\text{partition}(\text{PHASE}, \{\text{Env}\}, \{\text{Det}\}, \{\text{Cont}\}, \{\text{Pred}\})$

END

The sluice system: a refined model (cont.)

MACHINE

m_1

REFINES

m_0

SEES

c_1

VARIABLES

$door1, door2, pressure, phase, failure$

INVARIANT

$phase \in PHASE \wedge failure \in BOOL$

$failure = TRUE \Rightarrow phase = Cont$

$phase = Pred \Rightarrow failure = FALSE$

$phase = Env \Rightarrow failure = FALSE$

INITIALISATION

...

$phase, failure := Env, FALSE$

The sluice system: a refined model (cont.)

EVENTS

Environment =

WHEN *phase* = *Env* **THEN** *phase* := *Det* **END**

Detection =

WHEN *phase* = *Det* **THEN**

failure : \in *BOOL*

phase := *Cont*

END

open1 =

WHEN ... *phase* = *Cont* \wedge *failure* = *FALSE*

THEN ... *phase* := *Pred* **END**

close1 =

WHEN ... *phase* = *Cont* \wedge *failure* = *FALSE*

THEN ... *phase* := *Pred* **END**

...

The sluice system: a refined model (cont.)

...

```
<open2, close2, pressure_low, pressure_high modified similarly >
other_control =
  WHEN phase = Cont ∧ failure = FALSE
    THEN phase := Pred END
  stop =
    WHEN phase = Cont ∧ failure = TRUE
      THEN END
Prediction =
  WHEN phase = Pred
    THEN phase := Env END
END
```

Sluice example: a refined model (cont.)

- Here, *failure* abstractly models the unrecoverable system failure leading to the shutdown (stop) of the system
- Since the concrete detection mechanisms are still missing, failure detection is modelled non-deterministically as *failure* : \in *BOOL*
- The controller phase may contain other control actions (e.g., managing the pump or door motors), so we reserve a possibility to add these actions in the abstract event *other_control*
- The event *stop* can be also refined to include concrete system shutdown mechanisms

Sluice example: possible refinement plan

Five small incremental refinement steps:

- Introducing feedback loop of a control system (m1);
- Elaborating on the environment part and adding sensors (m2);
- Data refining failure modes (m3);
- Elaborating on error detection (m4);
- Introducing actuators and refining error prediction (m5).

Relations: introduction

- Modelling in the B Method based on sets – collections of elements of the same underlying type
- Often this is not enough: connections between elements of different types should be expressed
- Relations allow us to express more complicated interconnections and relationships formally
- Relations are often called many-to-many mappings

Relations (cont.)

- A relation R between sets S and T can be represented as a set of pairs (s, t) representing those elements of S and T that are related
- A pair is syntactically represented in Event-B as $(s \mapsto t)$ or $(s,,t)$ in ascii
- Mathematically, a relation between sets S and T is a member of $\mathbb{P}(S \times T)$, i.e., a subset of $S \times T$
- Reminder: $S \times T$ – all possible pairs from S and T
- Shorthand notation: $S \leftrightarrow T \equiv \mathbb{P}(S \times T)$
- In other words, $R \in S \leftrightarrow T$ is equivalent to $R \in \mathbb{P}(S \times T)$ or $R \subseteq S \times T$

Relations (cont.)

- Since a relation is just a special form of a set, all set operations are applicable to relations
- Example: a relation
 $owns_camera \in PERSON \leftrightarrow CAMERA$
- Initialisation:
 $owns_camera := \{Jonas \mapsto Canon, Vaidas \mapsto Nikon, Vaiva \mapsto Sony, Jonas \mapsto Sony, Sandra \mapsto Pentax\}$
- Checking for membership:
 $Jonas \mapsto Sony \in owns_camera$ (TRUE)
 $Vaiva \mapsto Canon \in owns_camera$ (FALSE)
- Similarly, \cup , \cap , \setminus , \subseteq , $card$, ... on relations

Relation domain and range

- The *domain* of a relation $R \in S \leftrightarrow T$ is the subset of elements of S that are related to something in T
- Relation domain (denoted as $\text{dom}(R)$) is defined by
$$\{x \mid x \in S \wedge \exists y. (y \in T \wedge (x, y) \in R)\}$$
- Example: $\text{dom}(\text{owns_camera}) = \{\text{Jonas}, \text{Vaidas}, \text{Vaiva}, \text{Sandra}\}$
- The *range* of a relation $R \in S \leftrightarrow T$ is the subset of elements of T that are related to something in S
- Relation range (denoted as $\text{ran}(R)$) is defined by
$$\{y \mid y \in T \wedge \exists x. (x \in S \wedge (x, y) \in R)\}$$
- Example: $\text{ran}(\text{owns_camera}) = \{\text{Canon}, \text{Nikon}, \text{Sony}, \text{Pentax}\}$

Relation filtering operations

Graphical notation, followed by the equivalent ascii notation:

$S \triangleleft R$	$S < R$	domain restriction
$S \trianglelefteq R$	$S << R$	domain subtraction
$R \triangleright S$	$R > S$	range restriction
$R \triangleright\triangleright S$	$R >> S$	range subtraction

Examples:

$$\{Jonas\} \triangleleft owns_camera = \{Jonas \mapsto Canon, Jonas \mapsto Sony\}$$

$$\{Mindaugas\} \triangleleft owns_camera = \emptyset$$

$$\{Jonas, Vaiva\} \trianglelefteq owns_camera = \{Vaidas \mapsto Nikon, Sandra \mapsto Pentax\}$$

$$\{Mindaugas\} \trianglelefteq owns_camera = owns_camera$$

$$owns_camera \triangleright \{Sony\} = \{Vaiva \mapsto Sony, Jonas \mapsto Sony\}$$

$$owns_camera \triangleright \{Sony, Canon, Pentax\} = \{Vaidas \mapsto Nikon\}$$

Homework: a hotel booking system

- The task: to create an Event-B system model within the Rodin platform for the given hotel reservation system requirements (the next two slides)
- The task has to be finished and presented within 3 weeks from today
- A finished Rodin project has to be exported (as a zip file) and submitted as your assignment solution from the course page in Moodle
- Separately, the solution should be personally "defended" during one of exercise sessions

Homework: a hotel booking system (requirements)

- ① The hotel booking system handles room reservation by customers;
- ② The system must have operations (events) for room reservation, cancellation, customer check-in, and customer check-out;
- ③ Only vacant hotel rooms can be reserved;
- ④ A reservation stores the information about the reserved room and the customer;
- ⑤ A reservation can be cancelled;
- ⑥ After cancellation, the previously reserved room becomes vacant;

Homework: a hotel booking system (requirements)

- ⑦ After a customer's check-in, the reserved room gets the status "occupied";
- ⑧ Each occupied room is reserved and associated with the same customer;
- ⑨ After a customer's check-out, the previously occupied room becomes vacant;
- ⑩ A vacant room cannot be at the same time considered as reserved or occupied by the system;
- ⑪ Once a reservation is cancelled or a customer checks out, all the information about the reservation is deleted from the system.

Homework: a hotel booking system (cont.)

Hints:

- Decide first on your static data in the context component (e.g., the needed abstract or enumerated sets)
- Model the vacant rooms as a set, while the reserved and occupied rooms as the corresponding relations;
- To remove the information from the relational variables, use the domain or range subtraction operations;
- Express the logical relationships between the vacant, reserved and occupied rooms as the respective system invariants.

Lecture 4: Outline

- Reminder: relations
- More operations on relations
- Example: a printer access system
- Total and partial functions (as a special kind of relations)
- Example: the vending machine revisited

Relations (reminder)

- A relation R between sets S and T can be represented as a set of pairs (s, t) representing those elements of S and T that are related
- Mathematically, a relation between sets S and T is a subset of $S \times T$ (or, equivalently, a member of $\mathbb{P}(S \times T)$)
- Note: $S \times T$ – all possible pair combinations between elements of S and T
- Note: $\mathbb{P}(S)$ – all possible subsets of S . Hence, $x \in \mathbb{P}(S)$ means that x is some subset of S , i.e., $x \subseteq S$
- $S \leftrightarrow T$ is a shorthand for $\mathbb{P}(S \times T)$

Relations (reminder)

- 3 equivalent statements saying that R is a relation between S and T :
 $R \in S \leftrightarrow T$, $R \in \mathbb{P}(S \times T)$, $R \subseteq S \times T$
- The *domain* of a relation $R \in S \leftrightarrow T$ is the subset of elements of S that are related to something in T
- Dually, the domain *range* of a relation $R \in S \leftrightarrow T$ is the subset of elements of T that are related to something in S
- We can filter relations, focusing on the relation domain (domain restriction $S \triangleleft R$ or domain subtraction $S \triangleleft\! R$) or the relation range (range restriction $R \triangleright S$ or range subtraction $R \triangleright\! S$)

Basic operations with relations

- Initialising (for $R \in S \leftrightarrow T$):

$$R := \{s_1 \mapsto t_1, s_2 \mapsto t_2, \dots\} \quad \text{or} \quad R := S \times \{t_0\}$$

where $s_i \mapsto t_i$ are constructed pairs from $S \leftrightarrow T$

- Adding, merging elements (using set operations):

$$R := R \cup \{s \mapsto t\} \quad \text{or} \quad R := R \cup Q$$

where Q is a relation of the same type, i.e., $Q \in S \leftrightarrow T$

- Checking membership or inclusion:

$$s \mapsto t \in R \quad \text{or} \quad R \subseteq Q$$

- Removing elements (filtering on the relation domain):

$$R := \{s_1, s_2, \dots\} \triangleleft R \quad \text{or} \quad R := \{s_1, s_2, \dots\} \trianglelefteq R$$

- Removing elements (filtering on the relation range):

$$R := R \triangleright \{t_1, t_2, \dots\} \quad \text{or} \quad R := R \triangleright \{t_1, t_2, \dots\}$$

More operations on relations

- Inverse relation R^{-1} (ascii R^\sim).

Includes all such $(x \mapsto y)$ that $(y \mapsto x) \in R$

- Example: for the relation

$\text{owns_camera} = \{\text{Jonas} \mapsto \text{Canon}, \text{Vaidas} \mapsto \text{Nikon},$
 $\text{Vaiva} \mapsto \text{Sony}, \text{Jonas} \mapsto \text{Sony}, \text{Sandra} \mapsto \text{Pentax}\}$

its inverse is

$\text{owns_camera}^{-1} = \{\text{Canon} \mapsto \text{Jonas}, \text{Nikon} \mapsto \text{Vaidas},$
 $\text{Sony} \mapsto \text{Vaiva}, \text{Sony} \mapsto \text{Jonas}, \text{Pentax} \mapsto \text{Sandra}\}$

- Relational image $R[S]$. Returns a subset of the relation range related to any element from the given set S

- Example: for the relation owns_camera defined above

$\text{owns_camera}[\{\text{Jonas}, \text{Vaiva}\}] = \{\text{Canon}, \text{Sony}\}$

More operations on relations (cont.)

- Relational composition $R_1; R_2$.

Composition of relations $R_1 \in S \leftrightarrow T$ and $R_2 \in T \leftrightarrow U$ is relation
 $\{(s \mapsto u) \mid \exists t. (s \mapsto t) \in R_1 \wedge (t \mapsto u) \in R_2\}$

- Example: Suppose we are given a relation between camera models and their brands $cmodels \in CMODEL \leftrightarrow CAMERA$,
for instance,
 $cmodels = \{350 \mapsto Canon, 60D \mapsto Canon, D5200 \mapsto Nikon, \dots\}$

Then the previously defined $owns_camera$ can be built as a result of a composition of the relations $owns_cmodel \in PERSON \leftrightarrow CMODEL$ and $cmodels \in CMODEL \leftrightarrow CAMERA$, i.e.,

$$owns_camera = owns_cmodel ; cmodels$$

More operations on relations (cont.)

- Relational overriding $R_1 \Leftarrow R_2$ (ascii $R_1 <+ R_2$).

Updating the relation R_1 by R_2 :

$$R_1 \Leftarrow R_2 = (\text{dom}(R_2) \Leftarrow R_1) \cup R_2$$

- Example:

$$\text{owns_camera} \Leftarrow \{\text{Jonas} \mapsto \text{Nikon}\}$$

Two pairs $\text{Jonas} \mapsto \text{Canon}$ and $\text{Jonas} \mapsto \text{Sony}$ are removed,
one pair $\text{Jonas} \mapsto \text{Nikon}$ is added

More operations on relations (cont.)

- Identity relation: $\text{id} \in S \leftrightarrow S$

The relation contains pairs $(s \leftrightarrow s)$ for all $s \in S$
(each element is only related to itself)

- Example: suppose we have the relation

$$\text{connected} \in \text{CITY} \leftrightarrow \text{CITY}$$

containing all the road connections between cities
(encoding a graph representation of roads between cities)

Then the requirements "Roads can only be between different cities"
can be formulated as

$$\text{connected} \cap \text{id} = \emptyset$$

Printer access example: requirements

- ① The system purpose is to manage user access to printers
- ② The dynamic information about which user has currently access to to which printer is stored by the system
- ③ This information can be updated by three operations: granting access, blocking access, or banning the user
- ④ In the first case, the user can be given access to a specific printer
- ⑤ In the second case, the user can be blocked from accessing a specific printer
- ⑥ In the last case, the user can be banned from using any printer

Printer access example: requirements

- ⑦ Each printer supports a number of printing options (like color or double side printing)
- ⑧ All printers support at least one printing option
- ⑨ Also, all options are supported by at least one printer
- ⑩ The information about all printer options is static and known beforehand
- ⑪ The user can send an enquiry about whether he or she has a possibility to use a specific printing option
- ⑫ The manager can send an enquiry about by all the users of a specific printer

Printer access example: context

CONTEXT

Access_ctx

SETS

USER

PRINTER

OPTION

PERMISSION

CONSTANTS

Options, Ok, No_permission

AXIOMS

$Options \in PRINTER \leftrightarrow OPTION$

$dom(Options) = PRINTER$

$ran(Options) = OPTION$

$partition(\{PERMISSION, \{Ok\}, \{NoAccess\}\})$

END

Printer access example: machine

MACHINE

Access_mch

SEES

Access_ctx

VARIABLES

access, last_query, pr_users

INVARIANT

$access \in USER \leftrightarrow PRINTER$

$last_query \in PERMISSION$

$pr_users \in \mathbb{P}(USERS)$

INITIALISATION

$access, pr_users := \emptyset, \emptyset$

$last_query := NoAccess$

Printer access example: machine

EVENTS

add_access =

ANY u, p

WHEN $u \in USER \wedge p \in PRINTER$

THEN $access := access \cup \{u \mapsto p\}$ **END**

block_access =

ANY u, p

WHEN $u \in USER \wedge p \in PRINTER \wedge (u \mapsto p) \in access$

THEN $access := access \setminus \{u \mapsto p\}$ **END**

ban_user =

ANY u

WHEN $u \in USER \wedge u \in \text{dom}(access)$

THEN $access := \{u\} \triangleleft access$ **END**

...

Printer access example: machine

...

```
perm_query_OK =  
  ANY u, o  
  WHEN u ∈ USER ∧ o ∈ OPTION  
    (u ↦ o) ∈ access; Options  
  THEN perm_query := Ok END  
perm_query_NOK =  
  ANY u, o  
  WHEN u ∈ USER ∧ o ∈ OPTION  
    (u ↦ o) ∉ access; Options  
  THEN perm_query := NoAccess END
```

...

Printer access example: machine

```
...
printer_query =
  ANY p
  WHEN  $p \in PRINTER$ 
  THEN  $pr\_users := access^{\sim}[\{p\}]$ 
  END
END
```

Functions

- Functions form a special class of relations that satisfy additional requirement: any element of the source set can be related to no more than 1 element of the target
- Functionality requirement mathematically:

$$\forall x, y, z. \ (x \mapsto y) \in R \wedge (x \mapsto z) \in R \Rightarrow y = z$$

- Any operation applicable to a relation or a set is also applicable to a function. For example, we can talk about the domain and the range of a function or a function as a set of pairs
- If f is a function, then $f(x)$ is the result of the function f for the argument x

Total and partial functions

- Functions are called *total* if their domain is the whole source set
Syntax: $f \in S \rightarrow T$ (or ascii $f : S \rightarrow T$)
where $\text{dom}(f) = S$ and $\text{ran}(f) \subseteq T$
- Example: the camera model relation is actually a total function
 $cmodels \in CMODEL \rightarrow CAMERA$
(any camera model identifier is uniquely associated with its brand)
- Functions are called *partial* if their domain is a subset of the source set
Syntax: $f \in S \leftrightarrow T$ (or ascii $f : S \rightarrow T$)
where $\text{dom}(f) \subseteq S$ and $\text{ran}(f) \subseteq T$
- Example: room reservation relation is actually a partial function
 $reserved \in ROOM \leftrightarrow CUSTOMER$
(each reserved room has the unique customer that served it, however, not all rooms must be reserved)

Arrays

- An array is a named, indexed collection of values of a given type.
- The array values can be accessed (read and updated) by using appropriate indexes.
- If we use $1..n$ (for some $n \in \mathbb{N}$) as our index set, then an array (containing elements of type S) can be modelled as a function from $1..n$ to S .
- In fact, any set can be used as the index set for arrays. Therefore, arrays can be usually modelled as total functions from S (index set) to T (the type of array values).

Functional (array) assignment

- The notation used to describe machine actions (i.e., assignments in the machine events) allows us to directly assign values to indexed elements of arrays:

$$a(i) := E$$

- This is just syntactic sugar for the following assignment:

$$a := a \Leftarrow \{(i \mapsto E)\}$$

- The assignment also works if a is modelled as a partial function. However, if we want to check/read values from such an array, we have to ensure/prove (by using the event guards and/or machine invariants) that the used index belongs to the function domain, i.e., $i \in \text{dom}(a)$

- $\text{reserved}(r) := \text{FALSE}$ OK!
- $nItems(j) := nItems(i) + 1$ only if $i \in \text{dom}(nItems)$

Vending machine example revisited

New (additional) requirements:

- ⑪ The system stores the constant information about the prices of served items
- ⑫ The payment operation is successful only if the supplied payment (credit) is sufficient to cover the price of the chosen item
- ⑬ The amount of the served items in the vending machine is limited
- ⑭ The system stores the current number of available items (for each item)

Vending machine example revisited (cont.)

- ⑯ The item selection operation is only possible if the number of available numbers is positive
- ⑯ After serving the product, the system decreases the corresponding number of available items of this kind
- ⑯ Initially, the system contains the pre-defined number of each item
- ⑯ The number of available items can be increased by loading some number of items of particular kind

Vending machine example: context

CONTEXT Vending_ctx

SETS

CHOICES

CONSTANTS

None

price

initial_amount

Items

AXIOMS

None \in *CHOICES*

price \in *CHOICES* $\rightarrow \mathbb{N}_1$

$\text{dom}(\text{price}) = \text{CHOICES} \setminus \{\text{None}\}$

initial_amount $\in \mathbb{N}_1$

Items \subseteq *CHOICES*

Items = *CHOICES* $\setminus \{\text{None}\}$

END

Vending machine example: machine

MACHINE

Vending_mch

SEES

Vending_ctx

VARIABLES

ready, choice, payed, served, nItems

INVARIANT

...

$nItems \in Items \rightarrow \mathbb{N}$

$choice \neq None \wedge served = FALSE \Rightarrow nItems(choice) > 0$

INITIALISATION

...

$nItems := Items \times \{initial_amount\}$

Vending machine example: machine (cont.)

EVENTS

choose =

ANY *choice* **WHERE** ... *nItems(choice)* > 0
THEN ... **END**

cancel = ...

pay = **ANY** *credit*

WHERE ... *credit* > *price(choice)*
THEN ... **END**

serve =

WHEN ...

THEN ... *nItems(choice)* := *nItems(choice)* - 1 **END**

...

Vending machine example: machine (cont.)

Additional event for refilling items:

```
...
add_items =
  ANY it, n
  WHERE
    ready = TRUE
    it ∈ Items
    n ∈ N1
  THEN
    nItems(it) := nItems(it) + n
  END
```

Lecture 5: Outline

- Reminder: functions
- Varieties of functions
- Service-oriented systems
- Example: a master component of a service-oriented system
- Homework: augmented hotel system

Functions (reminder)

- Functions form a special class of relations that satisfy additional requirement: any element of the source set can be related to no more than 1 element of the target
- Functionality requirement mathematically:

$$\forall x, y, z. \ (x \mapsto y) \in R \wedge (x \mapsto z) \in R \Rightarrow y = z$$

- Any operation applicable to a relation or a set is also applicable to a function. For example, we can talk about the domain and the range of a function or a function as a set of pairs
- If f is a function, then $f(x)$ is the result of the function f for the argument x

Total and partial functions (reminder)

- Functions are called *total* if their domain is the whole source set
Syntax: $f \in S \rightarrow T$ (or ascii $f : S \rightarrow T$)
where $\text{dom}(f) = S$ and $\text{ran}(f) \subseteq T$
- Functions are called *partial* if their domain is a subset of the source set
Syntax: $f \in S \leftrightarrow T$ (or ascii $f : S \rightarrow\!\!\!-\> T$)
where $\text{dom}(f) \subseteq S$ and $\text{ran}(f) \subseteq T$

Functional/array assignment (reminder)

- The notation used to describe machine actions (i.e., assignments in the machine events) allows us to directly assign values to indexed elements of arrays:

$$a(i) := E$$

- This is just syntactic sugar for the following assignment:

$$a := a \Leftarrow \{(i \mapsto E)\}$$

- The assignment also works if a is modelled as a partial function. However, if we want to check/read values from such an array, we have to ensure/prove (by using the event guards and/or machine invariants) that the used index belongs to the function domain, i.e., $i \in \text{dom}(a)$

- $\text{reserved}(r) := \text{FALSE}$ OK!
- $nItems(j) := nItems(i) + 1$ only if $i \in \text{dom}(nItems)$

Varieties of functions

Suppose we have a function f (from the source X to the target Y). Then it is called

Total function	\rightarrow	\rightarrow	if $dom(f) = X$, $ran(f) \subseteq Y$
Partial function	\rightarrow	$+ \rightarrow$	if $dom(f) \subseteq X$, $ran(f) \subseteq Y$
Total injection	\rightarrowtail	$> \rightarrow$	if $dom(f) = X$, $ran(f) \subseteq Y$ and one-to-one function
Partial injection	\rightarrowtail	$> + \rightarrow$	if $dom(f) \subseteq X$, $ran(f) \subseteq Y$ and one-to-one function
Total surjection	\twoheadrightarrow	$- \rightarrow$	if $dom(f) = X$, $ran(f) = Y$
Partial surjection	\rightarrowtail	$+ - \rightarrow$	if $dom(f) \subseteq X$, $ran(f) = Y$
(Total) Bijection	\rightarrowtail	$> - \rightarrow$	if $dom(f) = X$, $ran(f) = Y$ and one-to-one function

Varieties of functions: examples

- Injection: a function with 1-to-1 relationship between the source and target sets (e.g., an array without repeating elements)
- Example: $VU_id \in PERSON \rightarrowtail ID$

It is a partial injection: not all persons have a VU identification number, however, id is unique for each person

- Advantage: a reverse relation for an injection is also a function!
- Example: $VU_id^\sim \in ID \rightarrowtail PERSON$

A total injection this time

- Other examples:
 $Capital \in COUNTRY \rightarrow CITY$, and
 $Capital_of \in CITY \rightarrowtail COUNTRY$ (where $Capital_of = Capital^\sim$)

Varieties of functions: examples

- Surjection: a function that covers all the target set
- Example:

$\text{married} \in \text{WIFE} \twoheadrightarrow \text{HUSBAND}$

It is a total surjection

- Another example:
 $\text{Capital_of} \in \text{CITY} \twoheadrightarrow \text{COUNTRY}$

A partial surjection this time

Varieties of functions: examples

- Bijection: a total function that is both injection and surjection
- Example:

$\text{married} \in \text{WIFE} \rightarrowtail \text{HUSBAND}$

It is a bijection (in many countries)

- Bijections relate sets with the same power (length)
- Another example:

$\text{VU_account} \in \text{ID} \rightarrowtail \text{VU_ACCOUNT}$

A bijection: both sets are of the same length and one-to-one relationships in both directions

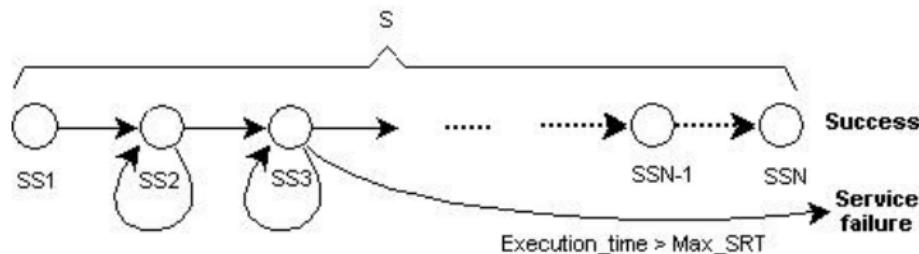
Telecommunicating and service-oriented systems

- Examples: telecommunication networks, service-oriented architectures, web services, cloud-based services
- Usually consists of multiple components that collectively provide a service to the service consumer (the external user or possibly other service provider)
- The components can be further partitioned into those that are responsible for service orchestration (management) or service execution
- Components for service orchestration: master components, service directors, service managers, (sometimes) frontend components
- Components for service execution: worker components, standalone components

Service decomposition

Often, a service request is decomposed and forwarded to different components (sub-service providers)

Service directors / master components are responsible for managing and controlling the whole service flow, forwarding requests to the respective components providing the required sub-services



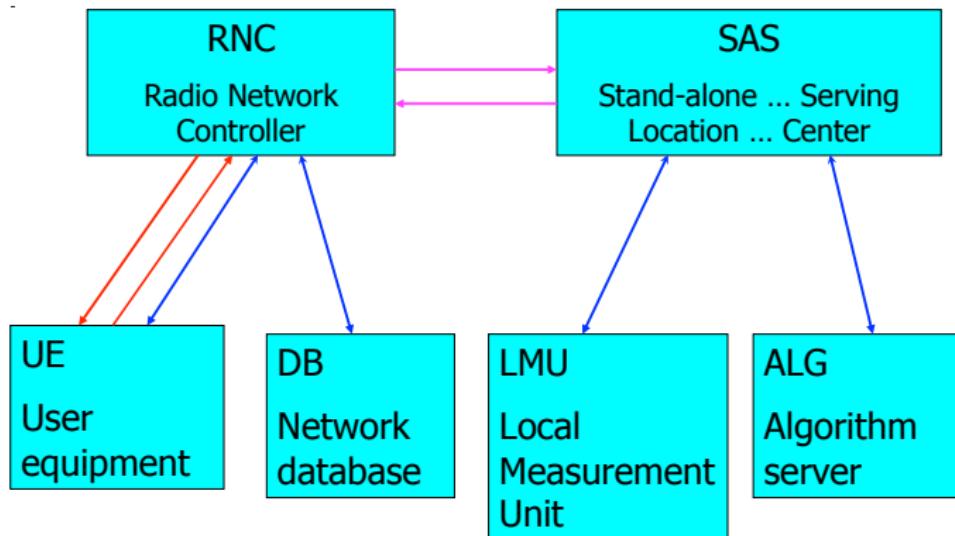
Telecommunicating and service-oriented systems (cont.)

- Components can be unavailable/busy or fail during execution
- A master component is responsible for monitoring the availability and failure status of its workers
- A master component can also incorporate fault tolerance mechanisms (e.g., retrying a service request, finding a replacement worker component, etc.)
- A master component can usually handle multiple service requests at once, while worker components are often executing a single service request

Positioning service (Nokia)

Typically, service-oriented systems have a layered architecture

Here, RNC (Radio Network Controller) – the external master (frontend) component, SAS – local service master component, while UE, DB, LMU, and ALG are employed as standalone / worker components



Simple example of a service-oriented system: requirements

- ① The system (master component) handles the incoming service requests, distributing them to the available worker components
- ② While being handled, the arrived service requests are stored in the input buffer
- ③ The input buffer has the pre-defined size that cannot be exceeded
- ④ Once handled, all the requests are stored in the output buffer (no size restrictions)
- ⑤ Each request has the handling status, which can be Waiting, Executing, Completed, or Failed
- ⑥ If the input buffer is full, a request is immediately transferred to the output buffer with the status Failed

Service-oriented system: requirements (cont.)

- ⑦ After an arrived request is added to the input buffer, it gets the status Waiting
- ⑧ Once a request from the input buffer is assigned to one of the available worker components, it gets the status Executing
- ⑨ Once a request is successfully handled by a worker component, it gets the status Completed and is transferred from the input buffer to the output buffer
- ⑩ A worker can fail to handle a request. Then the request gets the status Failed and is transferred from the input buffer to the output buffer

Service-oriented system: requirements (cont.)

- ⑪ Each request is associated with a specific service type
- ⑫ Each worker component is capable to handle a specific subset of service types. This knowledge is known beforehand and does not change during system execution
- ⑬ There is a number of active worker components at a particular moment. Any inactive worker can be activated and any active worker (not handling any requests) can be deactivated
- ⑭ Only an active worker component that is capable to handle the service type of a request can be assigned that request
- ⑮ A worker component can handle no more than one request at the moment

Service-oriented system: context

CONTEXT

Services_ctx

SETS

REQUEST

RTYPE

WORKER

STATUS

CONSTANTS

Waiting, Executing, Completed, Failed,

Req_type, Worker_types, buffer_size

...

END

AXIOMS

$\text{partition}(\text{STATUS}, \{\text{Waiting}\}, \{\text{Executing}\},$
 $\{\text{Completed}\}, \{\text{Failed}\})$

$\text{Req_type} \in \text{REQUEST} \rightarrow \text{RTYPE}$

$\text{Worker_types} \in \text{WORKER} \rightarrow \mathbb{P}_1(\text{RTYPE})$

$\text{buffer_size} \in \mathbb{N}_1$

END

Revealing a structure of abstract type elements

- Abstract types (like *REQUEST* or *WORKER*) are convenient to introduce standard notions of the modelled system
- If we need to reveal the internal structure of abstract type elements, we can introduce constant functions (like *Req_type*) in the context component to extract the necessary information from abstract type element
- Such functions are similar to a way object attributes (fields) are accessed in OOP
- Another possible function:
 $Req_priority \in REQUEST \rightarrow PRIORITY$

MACHINE

Services_mch

SEES

Services_ctx

VARIABLES

input, output, rstatus, active, assigned

INVARIANT

$\text{input} \in \mathbb{P}(\text{REQUEST})$

$\text{output} \in \mathbb{P}(\text{REQUEST})$

$\text{rstatus} \in \text{REQUEST} \leftrightarrow \text{STATUS}$

$\text{active} \in \mathbb{P}(\text{WORKER})$

$\text{dom}(\text{rstatus}) = \text{input} \cup \text{output}$

$\text{input} \cap \text{output} = \emptyset$

$\text{card}(\text{input}) \leq \text{buffer_size}$

...

Service-oriented system: machine (cont.)

$assigned \in active \rightarrow REQUEST$

$ran(assigned) \subseteq input$

$\forall r \cdot r \in ran(assigned) \Rightarrow rstatus(r) = Executing$

$\forall r \cdot r \in output \Rightarrow rstatus(r) \in \{Completed, Failed\}$

$\forall r \cdot r \in input \Rightarrow rstatus(r) \in \{Waiting, Executing\}$

INITIALISATION

$input, output := \emptyset, \emptyset$

$rstatus, assigned := \emptyset, \emptyset$

$active : \in \mathbb{P}(WORKERS)$

...

Service-oriented system: machine (cont.)

EVENTS

request_arrival = ANY r

WHERE $r \notin \text{dom}(rstatus) \wedge \text{card}(\text{input}) \leq \text{buffer_size}$

THEN

$\text{input} := \text{input} \cup \{r\}$

$rstatus(r) := \text{Waiting}$

END

request_rejection = ANY r

WHERE $r \notin \text{dom}(rstatus) \wedge \text{card}(\text{input}) = \text{buffer_size}$

THEN

$\text{output} := \text{output} \cup \{r\}$

$rstatus(r) := \text{Failed}$

END

...

Service-oriented system: machine (cont.)

...

request_assignment = ANY r, w

WHERE $r \in input \wedge w \in active \wedge rstatus(r) = Waiting$
 $Req_type(r) \in Worker_types(w)$

THEN

$assigned(w) := r$

$rstatus(r) := Executing$

END

request_completed = ANY r

WHERE $r \in ran(assigned)$

THEN

$output := output \cup \{r\}$

$input := input \setminus \{r\}$

$assigned := assigned \triangleright \{r\}$

$rstatus(r) := Completed$

END



Service-oriented system: machine (cont.)

...

request_failed = ANY r

WHERE $r \in ran(\text{assigned})$

THEN

$\text{output} := \text{output} \cup \{r\}$

$\text{input} := \text{input} \setminus \{r\}$

$\text{assigned} := \text{assigned} \triangleright \{r\}$

$rstatus(r) := Failed$

END

worker_activate = ANY w

WHERE $w \notin active$

THEN $active := active \cup \{w\}$ END

worker_deactivate = ANY w

WHERE $w \in active \wedge w \notin dom(\text{assigned})$

THEN $active := active \setminus \{w\}$ END

Invariants: summary

Different kinds/forms of invariants:

- ① Typing invariants:

$x \in S, y \subseteq S, r \in S \leftrightarrow T, f \in S \rightarrow T, \dots$

- ② Interrelationships between different variables (and constants) :

$input \cap output = \emptyset, f \subseteq r, dom(f) = y$

$card(input) \leq buffer_size$

The variable values can be used to directly define the typing of other variables:

$items \in POW(ITEMS), price \in items \rightarrow NAT$

Invariants: summary (cont.)

- ③ Conditional (local) invariants of the form
 $cond_1 \wedge \dots \wedge cond_k \Rightarrow property$

$served = \text{TRUE} \Rightarrow payed = \text{TRUE}$

$choice \neq \text{None} \wedge served = \text{FALSE} \Rightarrow nItems(choice) > 0$

In such a way, invariants can be "narrowed" down to specific points of system execution

- ④ Using quantifiers in invariants on more complex data structures:
 $\forall r \cdot r \in ran(\text{assigned}) \Rightarrow rstatus(r) = \text{Executing}$
 $\forall r \cdot r \in output \Rightarrow rstatus(r) \in \{\text{Completed}, \text{Failed}\}$

Homework: an extended hotel booking system (slightly changed requirements from the first homework)

- ① The hotel booking system handles room reservation by customers;
- ② The system must have operations (events) for room reservation, cancellation, customer check-in (with a reservation), customer check-in (without a reservation), and customer check-out;
- ③ Each reservation is stored by the system until it is cancelled or the reservation customer checks-in (with a reservation);
- ④ A reservation stores the information about the reserved room, the reserved dates, and the customer;
- ⑤ For any reservation, its dates cannot overlap with any other reservation dates for the same room;

Homework: an extended hotel booking system (requirements, cont.)

- ⑥ A reservation can be cancelled;
- ⑦ After cancellation, the stored reservation is removed from the system;
- ⑧ After a customer's check-in (with a reservation), the reserved room gets the status "occupied" and the stored reservation is removed from the system;
- ⑨ The system keeps the information about the occupied rooms, the customers, and the dates they are staying;
- ⑩ After a customer's check-in (with a reservation), the information from the room reservation is associated with (copied to) that of the occupied room;

Homework: an extended hotel booking system (requirements, cont.)

- ⑪ For any occupied room, its dates cannot overlap with any reservation dates for the same room;
- ⑫ A customer can check-in (without a reservation), if there is an unoccupied and unreserved room for the specified dates;
- ⑬ Once a customer checks-out, the information about the occupied room (the customer and dates) is removed from the system.

Homework: a hotel booking system (cont.)

Hints:

- Again, carefully define the necessary data structures in the model context
- Hotel reservations can be introduced as an abstract set, while the corresponding constant functions can be defined to extract necessary information (like the customer or dates) from them
- A similar or the same approach can be applied to model the info about the occupied rooms
- Dates can be also modelled as elements (subsets) of a pre-defined abstract set. Alternatively, a subset of natural numbers can be used
- The requirement 5 and 11 must become model invariants

Lecture 6: Outline

- Nondeterminism in Event-B
- Event-B model proof obligations
- Example: an electronic auction system

Nondeterminism in Event-B

- An Event-B machine models the system dynamics by describing its operations (events). Sometimes such a model of system dynamics is called a *state transition system*
- In their turn, the model events contain actions (state assignments), which describe system state changes/transitions
- Here a system state – a vector of the current system variable values
- The state changes can be deterministic (a typical state assignment with one possible outcome) or non-deterministic (one of from several/many state changes is possible)

Nondeterminism in Event-B (cont.)

- A simple example of a nondeterministic assignment:

$\text{res} : \{\text{Success}, \text{Failure}\}$

After this assignment, res can get any value from the set $\{\text{Success}, \text{Failure}\}$

- The most general form of non-deterministic assignment has the following syntax:

$x, y : | \text{Condition}(x, y, z, x', y')$

where x and y are the variables to be changed, z are the variables to be read only, and x' and y' are the new values for x and y

- Example:

$\text{time_left} : | \text{time_left}' < \text{time_left}$

any value from the interval $0.. \text{time_left} - 1$ can be assigned

Nondeterminism in Event-B (cont.)

- Declaring the event parameters/local variables in ANY clause – a special case of non-determinism

- A general event

ANY p **WHERE** $Cond(p, v)$ **THEN** $Actions$ **END**
is “executed” as a sequential composition

$p : | Cond(v, p'); Actions$

- There is no difference between having event parameters that are non-deterministically initialised by such implicit assignment or introducing local event variables with the same effect

- Example: $time_left : | time_left' < time_left$
can be rewritten as

ANY tt **WHERE** $tt \in \mathbb{N} \wedge tt < time_left$
THEN $time_left := tt$ **END**

Nondeterminism in Event-B (cont.)

- Machine events with overlapping guards – another special case of system non-determinism
- Any of such events can be chosen for execution, thus different event actions may be used to change the system state \Rightarrow the overall result is non-deterministic
- Example: Having two events

WHEN *component* \in active **THEN** *status* := Status1 **END**
and

WHEN *component* \in active **THEN** *status* := Status2 **END**

is equivalent to having one merged event

WHEN *component* \in active
THEN *status* : \in {Status1, Status2} **END**

Nondeterminism in Event-B (cont.)

- Non-determinism by machine events with overlapping guards can be used to express the situations where we have no knowledge or control over which event will occur first
- If the events operate over disjoint sets of variables (i.e., the updated variables are different), such overlapping events can also be used to model parallel calculations

Event-B proof obligations

- Event-B proof obligations (POs) define what it is to be proved for an Event-B model
- They are automatically generated (and in most cases also proved) by the Rodin platform
- Proof obligations – mathematical statements (theorems) about model correctness, consistency, and well-definedness
- There are several kinds of POs: invariant preservation, feasibility, well-definedness, theorem, etc.

Kinds of proof obligations: invariant preservation

- Proof obligations to ensure that each machine invariant is preserved by any machine event
- Proof obligations has the label INV
- Suppose we have a general form of an event:

```
ANY x  
WHERE G(s, c, v, x)  
THEN v :| Cond(s, c, v, x, v') END
```

where s and c are sets and constants from the context, v are machine variables, and x are event parameters or local variables.

Note that any standard assignment $x := \text{Exp}(x, y)$ is just equivalent to $x :| x' = \text{Exp}(x, y)$

Proof obligations: invariant preservation (cont.)

- Then, for such a general event, the invariant preservation POs are generated according to the following rule:

$A(s, c)$	Axioms and theorems
$Inv(s, c, v)$	Invariants and theorems
$G(s, c, v, x)$	Guards of the event
$Cond(s, c, v, x, v')$	The result condition of a non-det. assignment
\vdash	
$Inv_i(s, c, v')$	Specific invariant (in a post-state)

- Simplified form (without explicitly mentioning the sets and constants):
 $\vdash G(v, x) \wedge Inv(v) \wedge Cond(v, x, v') \Rightarrow Inv_i(v')$

For any possible state change, assuming all the invariants are true before the event and all the event guards are satisfied, the invariant in question remains true after the event is executed

Proof obligations: invariant preservation (cont.)

MACHINE Hotel

VARIABLES

..., reserved, occupied

INVARIANT

$reserved \in ROOM \rightarrow CLIENT$

$occupied \in ROOM \rightarrow CLIENT$

$occupied \subseteq reserved$

...

EVENTS

$new_reservation = \text{ANY } r, c$

WHERE $r \notin \text{dom}(reserved) \wedge c \in CLIENT$

THEN

$reserved(r) := c$

END

Proof obligations: invariant preservation (cont.)

- For the last specific invariant $\text{occupied} \subseteq \text{reserved}$, the generated proof obligation is

$\text{reserved} \in \text{ROOM} \rightarrow \text{CLIENT}$

$\text{occupied} \in \text{ROOM} \rightarrow \text{CLIENT}$

$\text{occupied} \subseteq \text{reserved}$

$r \notin \text{dom}(\text{reserved})$

$c \in \text{CLIENT}$

$\text{reserved}' = \text{reserved} \Leftarrow \{r \mapsto c\}$

\vdash

$\text{occupied} \subseteq \text{reserved}'$

- Or, in the simplified form:

$\text{occupied} \subseteq \text{reserved} \wedge r \notin \text{dom}(\text{reserved}) \wedge c \in \text{CLIENT} \Rightarrow$

$\text{occupied} \subseteq \text{reserved} \Leftarrow \{r \mapsto c\}$

Proof obligations: feasibility

- The purpose of feasibility POs is to ensure that a non-deterministic action is possible
- For a general event, the feasibility POs (labeled FIS) are generated according to the following rule:

 $A(s, c)$

Axioms and theorems

 $Inv(s, c, v)$

Invariants and theorems

 $G(s, c, v, x)$

Guards of the event

 \vdash $\exists v'. Cond(s, c, v, x, v')$

Nondeterministic action is possible

- Example: the event action

 $time_left : | time_left' < time_left$

(for $time_left \in \mathbb{N}$) is only provably feasible if $time_left > 0$

Proof obligations: well-definedness

- The purpose of well-definedness POs (labeled WD) is to ensure that all expressions are well-defined
- Examples of possibly ill-defined expressions:
applying a partial function to the argument that is not in the function domain (a special case – division by zero),
calculating the set cardinality (the number of elements) for a possibly infinite set, etc.
- In each separate case, the appropriate well-definedness condition is generated and asked to be proven
- A simple example: having an event guard
 $c = \text{reserved}(r)$
would automatically lead to a generated WD proof obligation:
 $r \in \text{dom}(\text{reserved})$

Proof obligations: theorems

- The theorem POs (labeled THM) are needed to ensure that the proposed context or machine theorems are provable
- You can propose a theorem by marking as such a specific axiom, invariant or guard that logically follow from their counterparts
- They seem to be redundant statements, however, once proven, they are automatically employed by the Rodin provers to prove other POs
- Often used as simpler intermediate statements (lemmas) that help in more complicated proofs

Other proof obligations

- If we refine/elaborate our models, we need to show that these refined models are consistent, i.e., they do not contradict the previously developed ones
- Rodin automatically generates additional proof obligations to ensure that
- Kinds of refinement proof obligations: simulation, guard strengthening, variants
- If refinement POs are successfully proven, all the verification results for the more abstract models are safely inherited

Simple example of a electronic auction system: requirements

- ① The system (master component) arranges and manages an electronic auction for selling items
- ② There is a number of registered sellers and buyers that are the users of an electronic action
- ③ Any seller can offer items to be sold
- ④ Once an item to be sold is submitted, the auction for this item starts.
The system can manage many items at once
- ⑤ During the auction, any buyer can make a bid for the item
- ⑥ A bid is only accepted if it exceeds the previous maximal bid for the same item

Electronic auction system: requirements (cont.)

- ⑦ The initial (minimal) item bid is given by a seller. By default, it is 0
- ⑧ There is the (pre-defined beforehand) maximal time for the auction to be completed
- ⑨ Once the maximal time runs out, the auction for this item is over and the maximal bid is declared a winner
- ⑩ The buyer that made the highest bid must pay for the item
- ⑪ Once the payment is made to the seller, the buyer receives the item
- ⑫ The system stores the information about what the current items are managed by the auction, which is the highest bid for each item, how much current time is left for each item, what the current items that were just paid for.

Electronic auction system: requirements (cont.)

- ⑬ All the information related to a specific item is erased from the system after the item is sold and the buyer receives the item
- ⑭ Additionally, the system stores the buyer and seller logs about which items have been sold and bought in the auction
- ⑮ The log information is accumulated (not erased)
- ⑯ The seller log is updated after the item payment is made
- ⑰ The buyer log is updated after the seller log
- ⑱ The information in both logs should be consistent for the items that are not currently being auctioned

Electronic auction system: context

CONTEXT

Auction_ctx

SETS

ITEM

SELLER

BUYER

CONSTANTS

Max_time

AXIOMS

$\text{Max_time} \in \mathbb{N}_1$

END

Electronic auction system: machine

MACHINE

Auction_mch

SEES

Auction_ctx

VARIABLES

$in_auction, time_left, item_seller, highest_bids,$
 $bids_buyer, paid_items, seller_log, buyer_log$

INVARIANT

$in_auction \in \mathbb{P}(ITEM)$

$time_left \in ITEM \rightarrow \mathbb{N}$

$dom(time_left) = in_auction$

$item_seller \in ITEM \rightarrow SELLER$

$dom(item_seller) = in_auction$

$highest_bids \in ITEM \rightarrow \mathbb{N}1$

$dom(highest_bids) \subseteq in_auction$

...



Electronic auction system: machine (cont.)

$bids_buyer \in ITEM \rightarrow BUYER$

$dom(bids_buyer) = dom(highest_bids)$

$paid_items \in \mathbb{P}(ITEM)$

$paid_items \subseteq in_auction$

$seller_log \in ITEM \rightarrow SELLER$

$buyer_log \in ITEM \rightarrow BUYER$

$dom(buyer_log) \subseteq dom(seller_log)$

$dom(seller_log) \cap in_auction \subseteq paid_items$

$dom(buyer_log) \cap in_auction \subseteq paid_items$

$\forall i. i \notin in_auction \Rightarrow (i \in dom(buyer_log) \Leftrightarrow i \in dom(seller_log))$

...

Electronic auction system: machine (cont.)

INITIALISATION

$in_auction, time_left, item_seller := \emptyset, \emptyset, \emptyset$
 $highest_bids, bids_buyer, paid_items := \emptyset, \emptyset, \emptyset$
 $seller_log, buyer_log := \emptyset, \emptyset$

EVENTS

$new_item = \text{ANY } i, s$

WHERE $i \in ITEM \wedge i \notin in_auction$
 $i \notin dom(buyer_log) \wedge s \in SELLER$

THEN

$in_auction := in_auction \cup \{i\}$
 $time_left(i) := Max_time$
 $item_seller(i) := s$

END

...

Electronic auction system: machine (cont.)

new_bid = ANY i, n, b

WHERE $i \in \text{dom}(\text{highest_bids}) \wedge n \in \mathbb{N}_1$

$n > \text{highest_bids}(i) \wedge b \in \text{BUYER}$

THEN

$\text{highest_bids}(i) := n$

$\text{bids_buyers}(i) := b$

END

new_bid_first_time = ANY i, n, b

WHERE $i \notin \text{dom}(\text{highest_bids}) \wedge i \in \text{in_auction}$

$n \in \mathbb{N}_1 \wedge b \in \text{BUYER}$

THEN

$\text{highest_bids}(i) := n$

$\text{bids_buyers}(i) := b$

END

...

Electronic auction system: machine (cont.)

...

payment = ANY i

WHERE $i \in in_auction \wedge i \in \text{dom}(highest_bids)$
 $time_left(i) = 0$

THEN

paid_items := *paid_items* $\cup \{i\}$

END

sell_confirmed = ANY i, s

WHERE $i \in paid_items \wedge s \in SELLER$

THEN *seller_log*(i) := s END

buy_confirmed = ANY i, b

WHERE $i \in paid_items \wedge b \in BUYER$

$i \in \text{dom}(\text{seller_log})$

THEN *buyer_log*(i) := b END

Electronic auction system: machine (cont.)

...

item_done = ANY *i*

WHERE $i \in in_auction \wedge i \in \text{dom}(buyer_log)$
 $i \in \text{dom}(seller_log)$

THEN

$in_auction := in_auction \setminus \{i\}$

$time_left, item_seller := \{i\} \triangleleft time_left, \{i\} \triangleleft item_seller$

$bids_buyer, paid_items := \{i\} \triangleleft bids_buyer, \{i\} \triangleleft paid_items$

$highest_bids := \{i\} \triangleleft highest_bids$

END

time_progress = ANY *i, new_time*

WHERE $i \in in_auction \wedge i \in time_left(i) > 0$

$new_time \in \mathbb{N} \wedge new_time < time_left(i)$

THEN

$time_left := new_time$

END

Time modelling

- Time is modelled by keeping track how much time is left for a particular item
- Time progress is specified by non-deterministic decrease of the stored time value (using a local variable *new_time*)
- Time decrease can be alternatively modelled as a non-deterministic assignment:

$$\text{time_left} : | \text{time_left}' < \text{time_left}$$

- Even though time progress is universal, we can model time progress separately for each auctioned item (because sellings of different items are independent)
- Such separation (as well as non-determinism) allows us avoid bigger complexity here

Lecture 7: Outline

- Multi-agent systems
- Agent roles, scope, and goals
- Adaptable systems
- Example: a multi-robotic system

Multi-agent systems

- Computer-based systems composed of multiple interacting intelligent/autonomous agents in some environment
- Agents: (typically) software agents, could be also robots or humans
- Essential characteristic: autonomy. Agents are (at least partially) independent, self-aware, autonomous
- Agents have their own local view (local knowledge, local perception) on the environment and base their decisions on this view

Multi-agent systems (cont.)

- Agents – decentralised and distributed entities, cooperating to achieve their individual goals
- Agents (typically) communicate asynchronously
- The characteristics of such systems:
 - have mobile elements (code, devices, data, services, users);
 - need to be context-aware;
 - are open (i.e., components can appear and disappear)

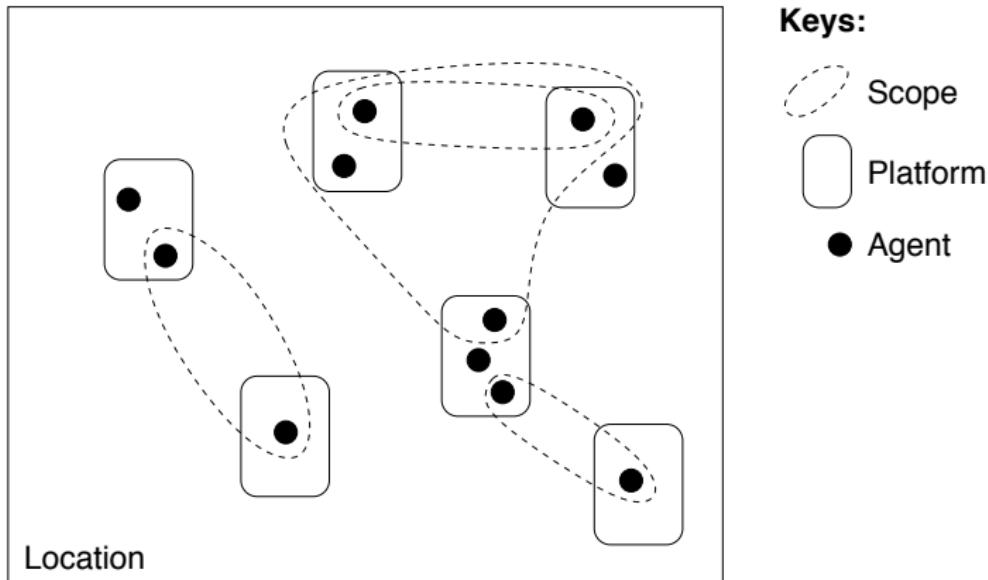
Multi-agent systems: agent roles

- Agent role: structuring unit of the agent functionality
- An agent role determines what it can do, in what joint activities it can participate, etc.
- Each agent has one or more roles associated with it
- May be used as a prerequisite before being involved in various agent cooperation activities, thus ensuring agent compatibility and interoperability
- Examples: *Buyer, Seller, Manager* (implicit) from the Auction system

Multi-agent systems: agent scope

- Scope – agent cooperation space (typically hosted at some location/server)
- Scope structures activity of agents at a specific location (physical or virtual)
- Scope provides isolation of several communicating agents, thus supporting error confinement and localised error recovery
- Examples: Moodle space for a specific course, “rooms” in a game server, teleconferencing, distant lecturing

Agent abstractions



The same device/component (platform) can host the same agent in different roles

Agent in a particular role collaborates/coordinates with other agents in a scope (if the scope conditions are satisfied)

Multi-agent systems: the scope entry and starting conditions

- Usually, an agent can join the scope activities if it satisfies the pre-defined conditions (e.g., it is in a specific role and the values of some its other attributes are sufficient)
- The scope itself has the conditions indicating when it becomes active. For instance, there should at least one agent in the role "Lecturer" and two agents of the role "Student" to activate the scope "Lecture"
- Since, in general, agents may join and leave a scope at will, the conditions may become broken and the scope activities are stopped/frozen

Multi-agent systems: perceptions and local knowledge

- Agents need to be self-aware of their environment/surroundings, including a general status of other collaborating agents
- Agent perceptions about their environment constitute their stored local knowledge
- Based on this (often incomplete) knowledge, agent make decisions about their next actions. Special rules for "calculating" the best possible action from current agent perceptions
- Crossing into the artificial intelligence (AI) territory here ...

Example: foraging ants as a scientific model

- Humans often blatantly copy/steal best solutions from the nature (naturally, the nature have had plenty time to really test them :))
- One scientific study of self-organising agents: foraging ants
- Ants rely on their perceptions (smells) to approximate distance from the nest, different food sources, and other ants
- Also, ants may leave pheromone traces to "point" other ants to the most promising direction
- We can formulate the rules that determine the next ant actions (which way to go) based on four perceptions and test the likelihood of finding the food or surviving

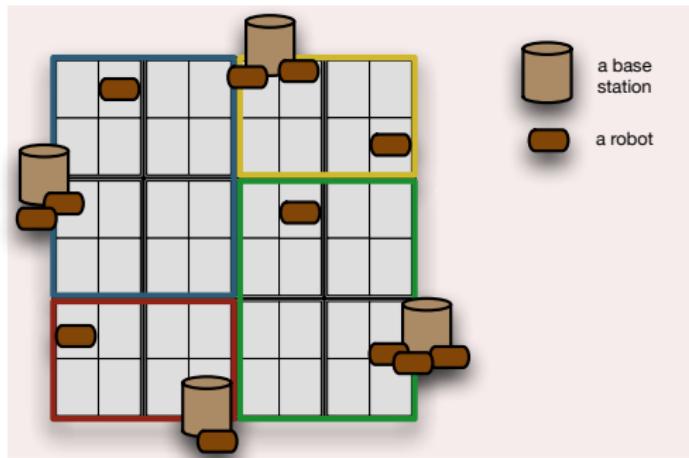
Multi-agent systems: goals

- Sometimes, the purpose of a multi-agent system can be expressed in terms of goals/mission that agent must try to accomplish
- Goals can be typically split into subgoals, forming goal hierarchies
- One of desired properties to be verified of such a multi-agent system is goal reachability

Adaptable systems

- Often, a multi-agent system is required to be flexible with respect to the changing environment ⇒ belong the class of adaptable systems
- A system must adapt to both negative (like agent failures or leaving) and positive (e.g., additional available resources) changes in the environment
- As a result, special system fault tolerance and dynamic reconfiguration mechanisms can be installed. Here, a system configuration can be understood as a particular distribution of agents in specific roles and environment resources
- Essentially, it is pre-defined agent cooperation scenarios to collectively recover from a bad situation or take full advantage of a good one

Example: a multi-robotic system



Cleaning an area by robots (workers), which are supervised by base stations. Robots are mobile, while base stations are stationary

Both types of agents are unreliable. In the case of failure, the system should dynamically reconfigure itself, if possible

Multi-robotic system: requirements

- ① The main goal of the multi-robotic system is to clean a given area
- ② The area to be cleaned is divided into a number of non-intersecting zones, which in turn are divided into a number of non-intersecting sectors. The number of sectors within different zones is the same
- ③ The area is considered as cleaned when each its zone is cleaned. A zone is considered as cleaned when each its sector is cleaned
- ④ The system involves two types of agents, base stations and robots, to achieve its main goal
- ⑤ Each active (i.e., non-failed) base station is associated with a number of the zones that it has a responsibility of cleaning
- ⑥ Each active base station is also associated with a number of robots that it is supervising

Multi-robotic system: requirements (cont.)

- ⑦ Each robot may be associated with a single sector that it is assigned to clean at the moment
- ⑧ The base stations are able to keep track of the cleaning process by storing the cleaning status of area sectors in their memory
- ⑨ A base station is able to communicate with other base stations and robots by sending messages
- ⑩ Communication between base stations can be used to 1) broadcast changes of the cleaning status of zone sectors, and 2) redistribute attached robots among the base stations
- ⑪ Communication from base stations to robots can be used to 1) assign cleaning tasks to the attached robots, and 2) change robot attachment to a different base station

Multi-robotic system: requirements (cont.)

- ⑫ Communication from robots to their base station can be used to report successful completion of an assignment
- ⑬ A base station may unrecoverably fail at any moment. An active base station has an ability to detect a failure of another base station
- ⑭ Once a base station fails, all its zones are considered as non-associated and all its robots as unattached
- ⑮ A robot may unrecoverably fail at any moment. A supervising base station has an ability to detect a failure of an attached robot
- ⑯ Only a robot, attached to some base station, can be given a cleaning assignment by this base station

Multi-robotic system: requirements (cont.)

- ⑯ Only non-cleaned sector can be assigned to a robot. A sector assigned to be cleaned cannot be already assigned to another robot
- ⑰ A cleaning assignment, communicated to a robot, should only concern cleaning of a sector belonging to one of the zones associated with the supervising base station
- ⑱ A robot cannot be given more than one assignment
- ⑲ A base station regularly updates its knowledge about cleaned sectors (after notifications from robots or broadcasts from other base stations)
- ⑳ A base station regularly communicates its local knowledge to the other base stations

Multi-robotic system: requirements (cont.)

- ㉒ If a base station has cleaned all its zones, its active robots may be reallocated under control of another base station
- ㉓ If a base station has no more active robots but still has sectors to clean, its zones and sectors may be reallocated under control of another base station
- ㉔ If a base station fails, its unassociated zones and unattached robots may be reallocated under control of another (active) base station

Multi-robotic system: context

CONTEXT

Robots_ctx

SETS

BSTATION

ROBOT

CONSTANTS

zone_number, sector_number, Zones, Sectors,
responsible_init, attached_init

AXIOMS

$\text{zone_number} \in \mathbb{N}_1 \wedge \text{sector_number} \in \mathbb{N}_1$

$\text{Zones} \in \mathbb{P}(\mathbb{N}_1) \wedge \text{Sectors} \in \mathbb{P}(\mathbb{N}_1)$

$\text{Zones} = 1..\text{zone_number}$

$\text{Sectors} = 1..\text{sector_number}$

$\text{responsible_init} \in \text{Zones} \rightarrow\!\!\! \rightarrow \text{BSTATION}$

$\text{attached_init} \in \text{ROBOT} \rightarrow\!\!\! \rightarrow \text{BSTATION}$

END

Multi-robotic system: notes

- Here *zone_number* and *sector_number* are the pre-defined numbers of zones and sectors. Essentially, they are global system parameters (any positive integer numbers)
- The types for all zones (*Zones*) and sectors (*Sectors*) are then constructed as the corresponding intervals of natural numbers. They are defined as constant sets
- The functions *responsible_init* and *attached_init* represent the initial values for which zones are supervised by which base stations and which robots belong to which base stations (i.e., initial system configuration)

Multi-robotic system: machine

MACHINE

Robots_mch

SEES

Robots_ctx

VARIABLES

*responsible, attached, assigned_zone, assigned_sector,
local_map, active_bs, active_rb*

INVARIANT

$\text{responsible} \in \text{Zones} \rightarrow \text{BSTATION}$

$\text{attached} \in \text{ROBOT} \rightarrow \text{BSTATION}$

$\text{assigned_zone} \in \text{ROBOT} \rightarrow \text{Zones}$

$\text{dom}(\text{assigned_zone}) \subseteq \text{dom}(\text{attached})$

$\text{ran}(\text{assigned_zone}) \subseteq \text{dom}(\text{responsible})$

$\forall r \cdot r \in \text{dom}(\text{assigned_zone}) \Rightarrow$

$\text{responsible}(\text{assigned_zone}(r)) = \text{attached}(r)$

...



Multi-robotic system: machine (cont.)

$\text{assigned_sector} \in ROBOT \rightarrow Sectors$

$\text{dom}(\text{assigned_zone}) = \text{dom}(\text{assigned_sector})$

$\text{active_bs} \in \mathbb{P}(BSTATION)$

$\text{ran}(\text{responsible}) = \text{active_bs}$

$\text{active_rb} \in \mathbb{P}(ROBOT)$

$\text{dom}(\text{attached}) \subseteq \text{active_rb}$

$\text{dom}(\text{assigned_zone}) \subseteq \text{active_rb}$

$\text{local_map} \in \text{active_bs} \rightarrow \mathbb{P}(Zones \times Sectors)$

...

Multi-robotic system: notes (cont.)

- The system keeps the information about the current configuration (*responsible*, *attached*) as well as goal/task assignment (*assigned_zone*, *assigned_sector*)
- Agents are unreliable \Rightarrow only the currently active agents (*active_bs*, *active_rb*) are taken into account
- *local_map* contains the perception of global situation (how many and which zones/sectors were cleaned) for each base station.
- For a particular base station, its perception about its own zones is more "fresh"/up-to-date than about the other zones

Multi-robotic system: machine (cont.)

INITIALISATION

```
responsible, attached := responsible_init, attached_init
assigned_zone, assigned_sector := Ø, Ø
local_map := BSTATION × {Ø}
active_bs, active_rb := BSTATION, ROBOT
```

EVENTS

attach_robot = ANY rb, bs

WHERE $rb \in active_rb \wedge bs \in active_bs$
 $rb \notin \text{dom}(\text{assigned})$

THEN

attached(rb) := bs

END

...

Multi-robotic system: machine (cont.)

assign = ANY bs, rb, z, s

WHERE $bs \in active_bs$

$rb \in dom(attached)$

$z \in dom(responsible)$

$s \in Sectors$

$attached(rb) = bs$

$responsible(z) = bs$

$rb \notin dom(assigned_zone)$

$(s \mapsto z) \notin local_map(bs)$

$\neg(\exists r \cdot r \in dom(assigned_zone) \wedge$

$assigned_zone(r) = z \wedge assigned_sector(r) = s)$

THEN

$assigned_zone(rb) := z$

$assigned_sector(rb) := s$

END

...



Multi-robotic system: notes (cont.)

- A lot of guards reflecting many restrictions for robot assignment, for instance:
 - Both base station and robot should be active;
 - Only a robot attached to a base station can be given assignment by this base station;
 - An assignment sector must be uncleaned;
 - An assigned sector must belong to a zone, for which a base station is responsible;
 - An agent cannot be given more than one assignment;
 - A sector cannot be already assigned to clean.

Multi-robotic system: machine (cont.)

```
sector_cleaned = ANY rb, bs
  WHERE rb ∈ dom(attached)
        bs ∈ active_bs
        attached(rb) = bs
        rb ∈ dom(assigned_zone)

  THEN
    local_map(bs) := local_map(bs) ∪
      {assigned_zone(rb) ↣ assigned_sector(rb)}
    assigned_zone := {rb} ◁ assigned_zone
    assigned_sector := {rb} ◁ assigned_sector

  END
```

...

Multi-robotic system: machine (cont.)

```
robot_fails = ANY rb
WHERE rb ∈ active_rb
THEN
    active_rb := active_rb \ {rb}
    attached := {rb} ⊲ attached
    assigned_zone := {rb} ⊲ assigned_zone
    assigned_sector := {rb} ⊲ assigned_sector
END
```

...

Multi-robotic system: machine (cont.)

```
bstation_fails = ANY bs, rbs
  WHERE bs ∈ active_bs
        rbs ⊆ ROBOT
        rbs = attached~ [{bs}]
  THEN
    active_bs := active_bs \ {bs}
    attached := attached ▷ {bs}
    responsible := responsible ▷ {bs}
    assigned_zone := rbs ⇣ assigned_zone
    assigned_sector := rbs ⇣ assigned_sector
    local_map := {bs} ⇣ local_map
  END
```

...

Multi-robotic system: notes (cont.)

- As a result of a base station failure, both zones and robots related to this base station become unassociated/"orphaned". Also, the information about assignments and the local map of the base station should be deleted
- How to calculate all the robots attached to a specific base station?
 $rbs = attached^{\sim} [\{bs\}]$
- These calculated robots (stored in a local variable rbs) are then used to filter out the assignment information

Multi-robotic system: machine (cont.)

```
reattach_robots = ANY rbs, bs1, bs2
WHERE rbs ⊆ ROBOT
    bs1 ∈ active_bs
    bs2 ∈ active_bs
    rbs ⊆ attached~ [{bs1}]
    rbs ∩ dom(assigned_zone) = ∅
THEN
    attached := attached ⇄ (rbs × {bs2})
END
```

...

Multi-robotic system: notes (cont.)

- The event describes a situation when a group of robots are transferred from one base station to another (because all the robots of some base station had failed or all sectors for which one base station is responsible are already cleaned)
- The transferred robots are a subset of all the attached robots:
 $rbs \subseteq attached^{\sim}[\{bs\}]$
- Only robots that are not involved in any current assignment can be transferred: $rbs \cap dom(\text{assigned_zone}) = \emptyset$

Multi-robotic system: machine (cont.)

update_map_from_others = ANY bs, map, zones

WHERE $bs \in active_bs$

$map \in \mathbb{P}(Zones \times Sectors)$

$zones \subseteq Zones$

$zones = responsible^\sim [\{bs\}]$

THEN

$local_map(bs) := map \Leftarrow (zones \triangleleft local_map(bs))$

END

...

Multi-robotic system: notes (cont.)

- The event describes updating the local map of a base station after receiving broadcasted update from some another base station
- Since the base station view on its own zones is more "fresh", only the information about sectors from other zones must be updated
- Here, the responsible zones are calculated as
$$\text{zones} = \text{responsible}^\sim [\{bs\}]$$
while the "fresher" part of the local map for the base station bs is
$$\text{zones} \triangleleft \text{local_map}(bs)$$
- Then
$$\text{local_map}(bs) := \text{map} \Leftarrow (\text{zones} \triangleleft \text{local_map}(bs))$$

overwrites the received data with a possibly more up-to-date information

Lecture 8: Outline

- Refinement in Event-B revisited
- Refinement proof obligations
- Example: a file transfer protocol
- Homework: a flight ticket system

Refinement in Event-B (reminder)

- A way to gradually develop formal models, elaborating on missing implementation details
- Allows to model the system at different abstraction levels, handle its complexity, and structure its requirements
- Consistency of model refinements is supported by the Rodin platform
- All the verification results (e.g., proved invariants) are inherited by the refined models
- Model refinement by refinement also facilitates automation by splitting/reducing overall complexity of model proofs

The notion of model refinement (cont.)

- Essential property: transitivity. Allows us to build a refinement chain of gradual development (unfolding) of the system;
- Mathematically:

$$\frac{M_1 \sqsubseteq M_2 \sqsubseteq \dots \sqsubseteq M_n}{M_1 \sqsubseteq M_n}$$

- All proven properties of more abstract models are preserved by refinement;
- Many formalisations based on the idea of model refinement, e.g., Refinement Calculus, VDM, Perfect Developer ...

Refinement of a context component

- For a context component, it is called *extension*;
- The keyword **EXTENDS** associates with an extended abstract context. All the definitions of the given context are transitively inherited
- Context extension allows
 - introducing new data structures (sets and constants), as well as
 - adding more constraints (axioms) for already defined ones.

Refinement of a machine component

- The keyword **REFINES** associates with a given abstract machine. All the invariants of the given machine are transitively inherited
- Moreover, the refined versions of inherited events have the clause **refines** *<old_event>*. All the remaining events are considered as new
- For a machine component, there are several possible kinds of refinement:
 - simple extension of an abstract model by new variables and events (*superposition refinement*);
 - constraining the behaviour of an abstract model (*refinement by reducing model non-determinism*);
 - replacing some abstract variables by their concrete counterparts (*data refinement*).

Superposition refinement

- Adding new variables and events;
- Reading and updating new variables in old event guards and actions;
- Interrelating new and old variables by new invariants;
- **Restriction:** the old variables cannot be updated in new events!
- Example of superposition refinement: in the sluice system (Lecture 3), the first refinement step introduced the controller phases (as a new variable). The old events became a part of the CONT phase, while other phases were modelled by new events

Refinement of non-determinism

- Focuses on the old (abstract) model events:
 - Strengthening the guards;
 - Providing several versions of the same event;
 - Refining non-deterministic actions (assignments).

```
evt =  
  WHERE g THEN  
    detected :∈ BOOL  
  END
```

```
evt1 refines evt =  
  WHERE g ∧ g' THEN  
    detected := TRUE  
  END
```

```
evt2 refines evt =  
  WHERE g ∧ g'' THEN  
    detected := FALSE  
  END
```

- Replacing some old variables by their concrete counterparts
- A part of concrete invariant, *gluing invariant*, describes the logical relationships between the old and new variables
- The gluing invariant is used in all proofs to show the correctness of such a replacement.

$$(\text{comm_failure} = \text{TRUE}) \Leftrightarrow (\text{msg_sent} = \text{FALSE} \vee (\text{msg_sent} = \text{TRUE} \wedge \text{msg_lost} = \text{TRUE}))$$

Data refinement: example

- In the sluice system abstract models (Lecture 3), system failures are modelled by the abstract boolean variable *failure*
- Once the information about door sensors is introduced, we can directly model detection of sensor failures
- The abstract variable *failure* then can be replaced (data refined) by the concrete variables *door1_sen_failure*, *door2_sen_failure*, and *other_failures*
- This is reflected by the added gluing invariant:

$$\text{failure} = \text{TRUE} \Leftrightarrow (\text{door1_sen_failure} = \text{TRUE} \vee \text{door2_sen_failure} = \text{TRUE} \vee \text{other_failures} = \text{TRUE})$$

Refinement proof obligations

- In addition to standard POs (like invariant preservation), a refined model should satisfy the following properties
 - guards of the old events are strengthened (or remain the same);
 - actions of the old events *simulate* those of the abstract ones – each refined model transition (execution step) is allowed by the abstract model;
 - the new events do not take over forever (do not get into infinite loop).
- In all POs, the gluing invariant is used to relate the old and new model states.

Kinds of proof obligations: guard strengthening for old events

- Proof obligations to ensure that refined events are called only in the situations that were abstractly modelled by the corresponding abstract counterparts
- Proof obligations has the label GRD
- Suppose we have a general form of an abstract event:

ANY x

WHERE $G_a(s, c, v, x)$

THEN $v : | Cond_a(s, c, v, x, v') END$

while the concrete its version is

ANY y

WHERE $G_c(s, c, w, y)$

THEN $v : | Cond_c(s, c, w, y, w') END$

Proof obligations: guard strengthening for old events (cont.)

- Then, for such events, the guard strengthening POs are generated according to the following rule:

$A(s, c)$ Axioms and theorems

$Inv_a(s, c, v)$ Abstract invariant

$Inv_c(s, c, v, w)$ Concrete invariant

$G_c(s, c, w, y)$ Concrete guards of the event

\vdash

$G_a(s, c, v, x)$ Abstract guards of the event

- Note that concrete invariant can refer to both abstract (v) and concrete (w) model variables

Kinds of proof obligations: simulation POs

- Proof obligations to ensure that the actions of a refined event are not contradictory with those of the corresponding abstract event
- Proof obligations has the label SIM
- Note that both GRD and SIM proof obligations concern with only the "old" events, defined in an abstract model and then refined in a refinement step

Proof obligations: simulation (cont.)

- Then, for such events, the simulation POs are generated according to the following rule:

$A(s, c)$ Axioms and theorems

$Inv_a(s, c, v)$ Abstract invariant

$Inv_c(s, c, v, w)$ Concrete invariant

$G_c(s, c, w, y)$ Concrete guards of the event

$Cond_c(s, c, w, y, w')$ The concrete action condition

\vdash

$Cond_a(s, c, v, x, v')$ The abstract action condition

Termination (convergence) of iterative events

- Sometimes we need to ensure that some iterative model events cannot take over forever (no infinite loops). Particular case: new events in a refined model
- This can be done by providing an expression (bounded from below with some minimal value) that is decreasing as a result of such event execution. Such an expression is called "variant"
- Decreasing a variant means that we definitely progressing towards "loop termination", since by definition such expression cannot be decreased indefinitely. Typically a variant is a natural number expression
- The corresponding generated POs to ensure that are labelled VAR

Termination (convergence) of iterative events (cont.)

- For such events, the termination POs are generated according to the following rule:

 $A(s, c)$

Axioms and theorems

 $Inv(s, c, v)$

Model invariant

 $G(s, c, v, x)$

Guards of the event

 $Cond(s, c, v, x, v')$

Nondeterministic action condition

⊤

 $n(s, c, v') < n(s, c, v)$ Decreasing of the variant

where n is a model variant

File transfer protocol: requirements

The example taken from the J.-R. Abrial's book (with slight modifications).

- ① The protocol (a version of two phase handshake protocol) ensures the copying of a file from one site to another one
- ② The file consists of a sequence of items
- ③ The file is sent item by item between the two sites
- ④ The sites exchange sending and receiving/acknowledgement operations through data channels
- ⑤ The exchanged information should be kept at minimum (only necessary)
- ⑥ Special procedures (like parity check) can be used to ensure data consistency

File transfer protocol: abstract context

CONTEXT

C_0

SETS

$DATA$

CONSTANTS

$fsize, f$

AXIOMS

$fsize \in \mathbb{N}_1$

$f \in 1..fsize \rightarrow DATA$

END

File transfer protocol: notes

- The file to be sent is constant during execution of the file transfer
- Therefore, we can model it as a constant function in the abstract model context
- Another known pre-defined constant is the file size – fsize

MACHINE

M0

SEES

C0

VARIABLES

g

INVARIANT

$g \in 1..fsize \rightarrow DATA$

INITIALISATION

$g := \emptyset$

...

EVENTS

```
receive =  
    status anticipated  
    WHEN  $g \neq f$   
    THEN  
         $g : \in 1..fsize \rightarrow DATA$   
    END  
  
final =  
    WHEN  $g = f$   
    THEN  
        skip  
    END
```

File transfer protocol: notes

- The received file (g) store some part of the original file (f). The variable is nondeterministically updated until the whole file is received
- We need to ensure that the receive event will eventually terminate. However, it is not enough information in the model to formulate a variant and prove termination (convergence)
- Compromise solution: the event gets the status "*anticipated*". It is a promise that, at some later refinement step, its convergence will be proved. In other words, its a postponed property proof
- *skip* in the event *final* stands for the absence of any actions

File transfer protocol: the first refinement

- We introduce a counter to receive the file piece by piece and refine the *receive* event
- Using the counter, we make it precise which portion of file is currently received and stored in g
- The counter also allows to formulate a model variant and turn the *receive* event into convergent
- The event guards also are rewritten in terms of the counter and the file size

File transfer protocol: first refinement

MACHINE

M1

REFINES M0

SEES

C0

VARIABLES

g, r

INVARIANT

$r \in 1..fsize + 1$

$g = (1..r - 1) \triangleleft f$

VARIANT

$fsize + 1 - r$

INITIALISATION

$g := \emptyset$

$r := 1$

...



File transfer protocol: first refinement (cont.)

EVENTS

receive =

status convergent

refines receive

WHEN $r \leq fsize$

THEN

$g(r) := f(r)$

$r := r + 1$

END

final =

refines final

WHEN $r = fsize + 1$

THEN

skip

END

File transfer protocol: second refinement

- Drawback: the receiver (in the *receive* event) has a direct access to the file f , which is situated at the sender site
- We need more distributed version of our protocol
- A model refinement introduces another event, *send*, changing the execution trace from

init → receive → ... → receive → final

to

init → send → receive → ... → send → receive → final

File transfer protocol: second refinement

MACHINE

M2

REFINES M1

SEES

C0

VARIABLES

g, r, s, d

INVARIANT

$s \in 1..fsize + 1$

$d \in DATA$

$s = r + 1 \Rightarrow d = f(r)$

$s \in r..r + 1$

VARIANT

$fsize + 1 - s$

...

File transfer protocol: notes

- A new counter, s , is introduced for the sender site
- Only the sender counter s and one data item are sent over the channel
- The counter value on the receiver site, r , is sent back as an acknowledgement
- We immediately prove that the new *send* event is convergent with the provided variant

File transfer protocol: second refinement (cont.)

INITIALISATION

```
 $g := \emptyset$ 
 $r, s := 1, 1$ 
 $d \in DATA$ 
```

EVENTS

```
receive =
    refines receive
    WHEN  $s = r + 1$ 
    THEN
         $g(r) := d$ 
         $r := r + 1$ 
    END
```

...

File transfer protocol: second refinement (cont.)

```
final =  
    refines final  
    WHEN  $r = fsize + 1$   
    THEN  
        skip  
    END  
  
send =  
    status convergent  
    WHEN  $s = r$   
         $s \neq fsize + 1$   
    THEN  
         $d := f(s)$   
         $s := s + 1$   
    END
```

File transfer protocol: third refinement

- It is not necessary to transmit the entire counters s and r on the data and acknowledgement channels
- The reasons: the counters only tested for equality, the only changes are increments by 1, the difference between s and r is at most 1
- Thus, the tests can be performed only on *parities* (last bits) of these counters

CONTEXT

C1

EXTENDS C0**CONSTANTS***parity***AXIOMS** $\text{parity} \in \mathbb{N}_1 \rightarrow \{0, 1\}$ $\text{parity}(1) = 1$ $\forall x. x \in \mathbb{N}_1 \Rightarrow \text{parity}(x + 1) = 1 - \text{parity}(x)$ **END**

File transfer protocol: third refinement

MACHINE

M3

REFINES M2

SEES

C1

VARIABLES

g, r, s, d, p, q

INVARIANT

$$p \in \{0, 1\}$$

$$q \in \{0, 1\}$$

$$p = \text{parity}(s)$$

$$q = \text{parity}(r)$$

$$p = q \Rightarrow s = r$$

$$p \neq q \Rightarrow s = r + 1$$

...

File transfer protocol: third refinement (cont.)

INITIALISATION

```
 $g := \emptyset$ 
 $r, s := 1, 1$ 
 $d \in DATA$ 
 $p, q := 1, 1$ 
```

EVENTS

```
receive =
    refines receive
    WHEN  $p \neq q$ 
    THEN
         $g(r) := d$ 
         $r := r + 1$ 
         $q := 1 - q$ 
    END
```

...

File transfer protocol: third refinement (cont.)

```
final =  
    refines final  
    WHEN  $r = fsize + 1$   
    THEN  
        skip  
    END  
  
send =  
    refines send  
    status convergent  
    WHEN  $p = q$   
         $s \neq fsize + 1$   
    THEN  
         $d := f(s)$   
         $s := s + 1$   
         $p := 1 - p$   
    END
```

File transfer protocol: notes

- The counter parity values are stored in the new variables p and q
- The checks in event guards are rewritten in terms of parities
- The parity and counter values are related in additional invariants
(needed to actually prove a refinement step, especially GRD POs)

Homework: flight ticket system (requirements)

- ① The purpose of the system is facilitate booking tickets to available flights between cities
- ② Each flight is associated with a pair of cities: the departure city and the destination city
- ③ The departure and destination cities for the same flight should be different
- ④ The same flight might occur on different dates (days)
- ⑤ The information about specific dates when the flight occurs is constant and is available in the system
- ⑥ There is a maximal number of plane seats, which can be different for each flight

Flight ticket system: requirements (cont.)

- ⑦ The system should keep track of a number of available tickets for a particular flight and a date
- ⑧ The system should provide an operation for booking (if possible) for a number of tickets for a particular flight on a particular date
- ⑨ The system should have an operation for cancelling of a number of tickets for a particular flight on a particular date
- ⑩ The system also keeps the information about the prices for a particular flight on specific dates
- ⑪ An additional booking operation should allow to book, if possible, two connecting flights (i.e., with the same intermediate city) for a pair of given cities, a specific date, and the maximal price of two flights
- ⑫ If there several pairs of possible connecting flights satisfying the criteria, the actual connecting flights are chosen non-deterministically

Lecture 9: Outline

- Automatic versus interactive theorem proving
- Interactive theorem provers: proof state, logical statements and goals
- Forward and backward (goal-oriented) proof styles
- Features of interactive proving in the Rodin platform
- Homework: a flight ticket system

The need for automation

- For even moderately sized programs (or their models), the complexity of correctness proofs becomes very high
- Assistance may be provided by a tool which records and maintains the proof as it is constructed step by step, and thus ensures its soundness
- Of course, the tool itself should satisfy basic requirements – be reliable and secure, i.e., perform only sound logical inferences
- Moreover, the logic of the tool should be expressive enough to represent all statements/expressions of our language or domain

Automatic vs interactive (theorem) proving

- Different levels of automation: fully automatic, fully interactive, something in between
- Fully automatic proving is more like a "push-button" technology
- The result of fully automatic proving is either confirmation of a successful proof of the given statement/property or a failure
- If a failure occurs, it is not clear, whether the result is wrong/false or the prover was not good enough to complete the proof
- Examples: "behind the scenes" application of automatic provers in Rodin or model checking in general

Automatic vs interactive (theorem) proving

- Fully automatic: (if succeeds) completely automatically produces the verification result (proved theorem). Everything happens as in "black box", no control over the process
- Fully automatic: usually possible for well-understood and precisely defined concrete classes of tasks/properties, when the proving goes according to the completely established pattern/procedure
- Proving of a particular task becomes just a concrete instantiation (of the parameters) of such a generic proving procedure
- Example: proving of a generic chain of B-Method/Event-B refinements for driverless train systems by Siemens Transportation France

Automatic vs interactive (theorem) proving

- Fully interactive: the user completely controls the proving process.
The computer system serves as checker/enforcer of the user proving commands
- Fully interactive: a proof becomes a program (in a specific language of proving commands). Proving scripts can be saved and reused
- Fully interactive: (advantage) very flexible, allows to experiment with any kinds of properties that can be formulated in the provers logic
- Fully interactive: (disadvantage) needs much more knowledge and expertise (more like "white box", where you can more effectively use it if you fully understand it)

Automatic vs interactive (theorem) proving

- Even in full-fledged interactive theorem provers, it is usually a mixture of bigger automatic steps (relying on integrated simplifiers, solvers, decision procedures) and smaller interactive steps (using only basic prover proof commands)
- Still, the main distinction of interactive proving comparing to the automatic one is that the current proof state is open and controlled by the user
- The user (by the available proof commands) modifies the proof state step-by-step until a new theorem is produced or the proof fails

Theorem provers

- There are general purpose interactive theorem provers (sometimes called proof assistants)
- Most actively developed or used: Isabelle, HOL (HOL-Light), Coq
- Can be used to define and reason about variety of domains as mathematical theories
- Functional programming languages (e.g., PolyML, OCaml) are used to manipulate proof state and write proof scripts
- The Rodin platform is not a general purpose prover and quite limited as an interactive theorem prover

Interactive theorem provers: hierarchical architecture

- Internal logic: the basic mathematical theories (like predicate calculus or set theory) that are used to define new notions and their properties
- Proof (meta) logic: a collection of axioms (accepted without proving) + the logical rules describing how new correct statements (theorems) can be produced from old ones
- Pre-defined or exported libraries of theorems and automated proving procedures (solvers, simplifiers, ...)

Theorems (logical sequents)

- Standard form of logical statement (a theorem if proved to be true):

$$H_1, H_2, \dots, H_3 \vdash C$$

- Here H_1, H_2, \dots, H_n are the hypotheses (assumptions). All logical (true or false) expressions
- C – the conclusion (logical expression)

- Example:

$$\text{even } x \vdash (x = 2) \vee \neg \text{prime } x$$

or, equivalently,

$$\vdash \forall x. \text{even } x \Rightarrow (x = 2) \vee \neg \text{prime } x$$

Different ways of proving a theorem

- We can construct a new theorem in different ways
- It can be a given axiom \Rightarrow it is a theorem right away (without a proof)
- It can be constructed from other theorems or axioms by combining them together in some way. Then it is called *forward proof*
- It can be stated as a goal (potential theorem) and then simplified/split until all the parts are immediately provable (from existing already theorems). Then it is called *backward proof*

Forward proof

- Produces a new theorem from existing ones by combining them in some way
- Proving commands: inference rules
- Inference rule – a parameterised function that takes theorems and other parameters and returns a new theorem

Forward proof: examples

- Combining two theorems by conjunction: for

$$H_{11}, H_{12}, \dots, H_n \vdash C_1$$

and

$$H_{21}, H_{22}, \dots, H_m \vdash C_2$$

the inference rule CONJ produces

$$H_{11}, H_{12}, \dots, H_n, H_{21}, H_{22}, \dots, H_m \vdash C_1 \wedge C_2$$

- Merging two theorems into one

Forward proof: examples

- Specialising an universally quantified theorem: for

$$H_1, H_2, \dots, H_n \vdash \forall x. C[x]$$

where $x \in \mathbb{N}$, the inference rule SPEC 0 produces

$$H_1, H_2, \dots, H_n \vdash C[0]$$

with all x replaced with 0

- Generates a theorem for a specific case from a general (universally quantified) case

Forward proof: examples

- Modus ponens

$$H_1, H_2, \dots, H_n \vdash P \Rightarrow Q$$

and

$$H_1, H_2, \dots, H_m \vdash P$$

where $m \leq n$, the inference rule MP produces

$$H_1, H_2, \dots, H_n \vdash Q$$

- Combines an implicative theorem with a theorem that proves its antecedent

Backward (goal oriented) proof

- Starts with potentially true statement (goal)
- The same standard form of logical statement (a theorem if proved to be true):

$$H_1, H_2, \dots, H_3 \rightarrow? C$$

- Proof commands: tactics. Used to modify/simplify/split the current goal until it can be proved in one step
- Tactics are functions that take a goal (logical sequent) + (possibly) some other parameters and produce a list of goals that are sufficient to prove the original one

Backward proof: examples

- Often, dual to the corresponding inference rules
- Splitting a conjunctive goal into two: for

$$H_1, H_2, \dots, H_n \dashv? C_1 \wedge C_2$$

the tactic CONJ_TAC produces

$$H_1, H_2, \dots, H_n \dashv? C_1$$

and

$$H_1, H_2, \dots, H_n \dashv? C_2$$

- Splitting a goal into two subgoals

Backward proof: examples

- Specialising an existentially quantified goal: for

$$H_1, H_2, \dots, H_n \dashv? \exists x. C[x]$$

where $x \in \mathbb{N}$, the inference rule EXISTS_TAC 0 produces

$$H_1, H_2, \dots, H_n \dashv? C[0]$$

with all x replaced with 0

- Generates a goal for a specific suggested case from a general (existentially quantified) case

Backward proof: examples

- Splitting a goal into two, by considering alternative cases: for

$$H_1, H_2, \dots, H_n \dashv? C$$

the tactic `BOOL_CASES_TAC "x=0"` produces

$$H_1, H_2, \dots, H_n, x = 0 \dashv? C$$

and

$$H_1, H_2, \dots, H_n, x \neq 0 \dashv? C$$

- Splitting a goal into two subgoals, by adding extra assumptions for the cases considered

Working with the proof state

- The proof state contains, at least, the current unproved subgoals
- Usually it also stores the proof tree, with all previous proof branches and proof commands
- Applying a tactic in interactive proof modifies the proof state
- With the whole proof tree available, it is possible backtrack to some previous state and try again

Mixing the backward and forward proof styles

- In backward proof, we can combine both proving styles
- For instance, we can add new assumptions if these assumptions are based on proved theorems
- If the existing assumptions (considered as theorems) can produce new theorems by applying some inference rules, the resulting theorems can be added as new assumptions
- Example: if a universally quantified assumption $\forall x. C[x]$ can be specified for concrete x value, say x_0 , then the new assumption $C[x_0]$ can be added

Other more advanced stuff

- **Tacticals:** merging tactics together
- **Simplifiers, rewriters, decision procedures, model checkers**
- **Other proving styles:** Mizar, window inference
- **A big number of contributions:** many theories, libraries, automated tools

Interactive proving in the Rodin platform

- Quite limited comparing to full fledged interactive theorem provers such as HOL, Isabelle, or Coq
- Rodin mostly relies on already integrated or possibly added as plug-ins automatic solvers (predicate provers pp and npp, mini-lemma prover/simplifier ml, the plug-in for integrating external SMT solvers, the plug-in for bridging with Isabelle)
- Interactive proving based on applying proof commands (tactics) for working with an unsolved current goal

Interactive proving in the Rodin platform (cont.)

- A separate Eclipse perspective – Proving
- Shows the current goal, hypothesis, proof tree, proof information in subwindows
- Provides access to automatic Rodin (installed) provers as well as simple proof tactics
- Proofs can be "pruned" in the Proof Tree view, essentially backtracking to some previous proof step

Interactive proving in the Rodin platform (cont.)

Some tactics:

- lasso – catching additional hypothesis for all the identifiers (names) in the goal
- ah (add hypothesis) – stating a simple property, which starts a separate proof. If successfully proven, the property is then added as extra hypothesis. An example of forward proof
- dc (deduce cases) – splits the proof for separate cases of the given expression (like the *BOOL_CASES_TAC* before)
- If the expression is boolean, two cases (it is true or false) are separately assumed and added as extra hypotheses. If the expression is an enumerated set, all different set values are assumed, splitting the proof into the respective number of cases

Interactive proving in the Rodin platform (cont.)

More tactics are hidden under red-underlined symbols in the hypotheses or the goal:

- Underlined \forall symbol in a hypothesis. If clicked, allows to instantiate the hypothesis by submitting concrete values for the quantified variable (forward proof on a hypothesis)
- Similarly, underlined \exists symbol in a goal. If clicked, allows to instantiate the goal by suggesting a concrete value for quantified variable
- Underlined $=$ in an equational hypothesis. If clicked, allows to rewrite the goal according to the hypothesis equation
- Underlined \subset or \subseteq . If clicked allows to replace, e.g., a set inclusion $s1 \subseteq s2$ with the equivalent expression $\forall x. x \in s1 \Rightarrow x \in s2$
- ...

Customising Event-B tactics in Rodin

- Rodin preferences: Event-B: Sequent Prover : Auto/Post Tactic
- Allows to combine simple tactics and automatic provers, saving as separate profiles
- Such saved scripts can be chosen to apply instead of the installed default simplifying/proving procedures

Lecture 10: Outline

- Case study: a lift system
- The lift system: requirements
- The lift system: models (first developed by J.-R. Abrial)
- The lift system: deconstructing a formalisation

A case study: a lift system (requirements)

- ① A lift system consists of a non-zero number of lifts installed in a building
- ② The building has a number of floors (more than one)
- ③ A lift can move between floors in one of two directions: up (unless it is already at the top floor) or down (unless it is already at the ground floor)
- ④ Each of floors (except ground and top) has two buttons, one to request an up-lift and one to request a down-lift
- ⑤ The ground and top floors have one button to request an up-lift and a down-lift respectively
- ⑥ When pushed, the buttons remain illuminated until a lift, traveling in the desired direction, visits a floor

The lift system requirements (cont.)

- ⑦ If both floor buttons are pushed, only respective one is cancelled after a lift visits a floor
- ⑧ Each lift has a set of buttons, one button for each floor, to request a stop at that floor
- ⑨ The buttons are illuminated when pressed and cause a lift to visit the corresponding floor (unless the lift is already at that floor)
- ⑩ The illumination is cancelled when the corresponding floor is visited
- ⑪ Each lift can either moving or stopped at some floor
- ⑫ A moving lift stops at a floor if it passes it and either (i) there are internal lift requests to stop at that floor, or (ii) there are floor requests on that floor to travel in the current lift direction

The lift system requirements (cont.)

- ⑬ When a lift stops at a floor, the floor is considered visited and all the related requests (both within the lift and on the floor) are cancelled
- ⑭ When a lift has no requests to service, it should remain at its final destination and await further requests
- ⑮ All requests for lifts from floors must be serviced eventually, with all floors given equal priority
- ⑯ All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel
- ⑰ Each lift keeps its current moving direction while there are floors with requests to visit at that direction
- ⑱ If there are no such requests to continue on its current direction and there are servicing requests in the opposite direction, a stopped lift should change its direction to the opposite one

The lift system requirements (cont.)

- ⑯ The time is not considered in the first system prototype
- ⑰ Once a lift stops at some particular floor, all people inside a lift that requested that floor are considered immediately out
- ⑱ Moreover, all people waiting on the floor to go to the lift current direction, are considered immediately in

Lift system: context

CONTEXT

Lift_ctx

SETS

LIFT

DIRECTION

CONSTANTS

up, dn, ground, top, FLOOR,

attracted_up, attracted_dn,

can_continue_up, can_continue_dn

AXIOMS

partition(DIRECTION, {up}, {dn})

ground ∈ N

top ∈ N

ground < top

FLOOR = ground .. top

...



Lift system: notes

- All three ways to introduce your own type are demonstrated here: as an abstract set, as an enumerated set, as a set constant
- The data type *FLOOR* is defined as a set constant
- It is a constructed type (set), defined the number interval between two other constants *ground* and *top*
- The constants *attracted_up*, *attracted_dn*, *can_continue_up*, *can_continue_dn* will be explained later

Lift system: machine

MACHINE

Lift_mch

SEES

Lift_ctx

VARIABLES

moving, floor, dir, in, out

INVARIANT

$\text{moving} \subseteq \text{LIFT}$

$\text{floor} \in \text{LIFT} \rightarrow \text{FLOOR}$

$\text{dir} \in \text{LIFT} \rightarrow \text{DIRECTION}$

$\text{in} \in \text{FLOOR} \leftrightarrow \text{DIRECTION}$

$\text{out} \in \text{LIFT} \leftrightarrow \text{FLOOR}$

...

Lift system: notes

- $\text{floor} \in \text{LIFT} \rightarrow \text{FLOOR}$
 $\text{dir} \in \text{LIFT} \rightarrow \text DIRECTION$
return the current floor and direction for a given lift
- $\text{in} \in \text{FLOOR} \leftrightarrow \text{DIRECTION}$
contains floor requests, i.e., the requests to get in a lift going in a specific direction

Example: $5 \mapsto dn \in \text{in}$ means that there people on the fifth floor wanting to go down
- $\text{out} \in \text{LIFT} \leftrightarrow \text{FLOOR}$
contains lift requests to go to a particular floor, i.e., the requests to get out a lift

Example: $/1 \mapsto 4 \in \text{out}$ means that there people in the lift /1 wanting to go to 4th floor

Lift system: machine (cont.)

ground $\mapsto dn \notin in$

top $\mapsto up \notin in$

moving $\Leftarrow (out \cap floor) = \emptyset$

$\forall l \cdot l \notin moving \Rightarrow floor(l) \mapsto dir(l) \notin in$

INITIALISATION

moving, in, out := $\emptyset, \emptyset, \emptyset$

floor := *LIFT* $\times \{ground\}$

dir := *LIFT* $\times \{up\}$

...

Lift system: notes

- The first two invariants (on the last slide) constrain floor requests: it is not possible to request an down-lift from the ground floor or an up-lift from the top floor
- The last two invariants formalise the following modelling requirements:

Once a lift stops at some particular floor, all the related floor and lift requests are cancelled. Moreover, all people inside a lift that requested that floor are considered immediately out. Moreover, all people waiting on the floor to go to the lift current direction, are considered immediately in

Deconstructing invariants

- Once a lift stops at some particular floor, all people inside a lift that requested that floor are immediately out (and their lift requests are cancelled)

$$moving \triangleleft (out \cap floor) = \emptyset$$

- Deconstructing:

$$out \cap floor$$

set of pairs $LIFT \times FLOOR$ such that the lift is on some particular floor and there people requested to move to that floor

$$moving \triangleleft (out \cap floor)$$

the moving lifts are not considered – the domain subtraction operation keeps only such pairs where lifts are stopped

Deconstructing invariants (cont.)

- $\text{moving} \Leftrightarrow (\text{out} \cap \text{floor}) = \emptyset$
there no such pairs left

It means that are no such requests to get out on a floor once a lift stopped on that floor

Which can only mean that,
for any stopped lift, all people inside a lift that requested that floor are considered immediately out (and their lift requests are cancelled)

Deconstructing invariants (cont.)

- "... Moreover, all people waiting on the floor to go to the lift current direction, are immediately in (and their floor requests are cancelled)"

$$\forall I \cdot I \notin \text{moving} \Rightarrow \text{floor}(I) \mapsto \text{dir}(I) \notin \text{in}$$

- Deconstructing:

For any stopped lift on some floor, there no requests from that floor to go in the direction the lift is going

⇒

all people waiting on the floor to go to the lift current direction, are already in (and their floor requests are cancelled)

Lift system: machine (cont.)

```
request_lift_from_floor =  
  ANY f, d  
  WHEN f ∈ FLOOR ∧ d ∈ DIRECTION  
    f ↪ d ≠ ground ↪ dn  
    f ↪ d ≠ top ↪ up  
    ∀l. l ∉ moving ⇒ ¬(floor(l) = f ∧ dir(l) = d)  
  THEN  
    in := in ∪ {f ↪ d}  
  END
```

...

Lift system: notes

- The last guard is needed to prove the formulated invariant:

$$\forall l \cdot l \notin moving \Rightarrow floor(l) \mapsto dir(l) \notin in$$

- Essentially stating the necessary condition: for any non-moving (stopped) lift on some floor, a lift request from this floor is only allowed if a lift in the desired direction is not already on this floor

Lift system: machine (cont.)

EVENTS

```
request_floor_from_lift =  
  ANY I, f  
  WHEN I ∈ LIFT ∧ f ∈ FLOOR  
    I ∉ moving ⇒ floor(I) ≠ f  
  THEN  
    out := out ∪ {I ↦ f}  
  END
```

...

Lift system: notes

- Again, the last guard is needed to prove the formulated invariant:

$$moving \Leftarrow (out \cap floor) = \emptyset$$

- Stating the necessary condition: for any non-moving (stopped) lift on some floor, a floor request from this lift is only allowed if the lift is not already on this floor

Lift system: notes

- In the remaining operations, we will often need to check whether there is a reason for a particular lift to go up or down
- In other words, whether there are the floors above or below where the users have requested to get in or out the lift
- We call this property as "*a lift is attracted up (down)*"
- We can formalise "*attracted up*" as follows

$$(dom(in) \cup out[\{l\}]) \cap ((floor(l) + 1) .. top) \neq \emptyset$$

Deconstruction a property

- $\text{dom}(\text{in})$
the floors where the users have requested a lift
- $\text{out}[\{l\}]$
(using the relational image operator) the floors to which the lift is requested to go
- $(\text{floor}(l) + 1) \dots \text{top}$
the floors above
- All together,
 $(\text{dom}(\text{in}) \cup \text{out}[\{l\}]) \cap ((\text{floor}(l) + 1) \dots \text{top}) \neq \emptyset$

"*There are the floors above where the users have requested to get in or out the lift*"

Deconstruction a property (cont.)

- Similarly, "attracted down" can be formalised as follows

$$(dom(in) \cup out[\{l\}]) \cap (ground .. (floor(l) - 1)) \neq \emptyset$$

"There are the floors below where the users have requested to get in or out the lift"

- We are going to check these property or their negations as well as use them as a part of more complex definitions quite often. How we could introduce the respective shorthands?
- Two solutions:
 - install the Theory extension (extra time for learning needed);
 - define the respective higher-order functions *attracted_up* and *attracted_dn* in the model context and use them in the machine when needed.

Lift system: context (cont.)

2 extra axioms are added to the context for each definition: one for giving the type for such a higher-order function, the other one for defining its returned values

$$\begin{aligned} \text{attracted_up} &\in \text{LIFT} \times (\text{LIFT} \rightarrow \text{FLOOR}) \times (\text{FLOOR} \leftrightarrow \text{DIRECTION}) \\ &\quad \times (\text{LIFT} \leftrightarrow \text{FLOOR}) \rightarrow \text{BOOL} \\ \forall I, \text{floor}, \text{in}, \text{out}. \quad \text{attracted_up}(I \mapsto \text{floor} \mapsto \text{in} \mapsto \text{out}) &= \\ &\quad \text{bool}(\text{dom}(\text{in}) \cup \text{out}[\{I\}]) \cap ((\text{floor}(I) + 1) .. \text{top}) \neq \emptyset \end{aligned}$$
$$\begin{aligned} \text{attracted_dn} &\in \text{LIFT} \times (\text{LIFT} \rightarrow \text{FLOOR}) \times (\text{FLOOR} \leftrightarrow \text{DIRECTION}) \\ &\quad \times (\text{LIFT} \leftrightarrow \text{FLOOR}) \rightarrow \text{BOOL} \\ \forall I, \text{floor}, \text{in}, \text{out}. \quad \text{attracted_dn}(I \mapsto \text{floor} \mapsto \text{in} \mapsto \text{out}) &= \\ &\quad \text{bool}(\text{dom}(\text{in}) \cup \text{out}[\{I\}]) \cap (\text{ground} .. (\text{floor}(I) - 1)) \neq \emptyset \end{aligned}$$

Lift system: notes

- The functions take as parameters the given lift as well as the functions containing info about lift floor, direction, and the requests to get in or out, and evaluate all them together them as *TRUE* or *FALSE*
- In addition to the typing axiom, the other axiom gives the exact definition of such evaluation
- The operator *bool* directly returns *TRUE* or *FALSE* for the given logical expression (predicate)
- It allows to replace two axioms:
 $\forall x, y, \dots cond \Rightarrow f(x \mapsto y \mapsto \dots) = TRUE$
and
 $\forall x, y, \dots not\ cond \Rightarrow f(x \mapsto y \mapsto \dots) = FALSE$
- Now we can directly use the introduced functions in machine event guards

Lift system: machine (cont.)

```
depart_up =  
ANY I  
WHEN  
    I ∉ moving  
    dir(I) = up  
    attracted_up(I ↪ floor ↪ in ↪ out) = TRUE  
THEN  
    moving := moving ∪ {I}  
    floor(I) := floor(I) + 1  
END
```

...

Lift system: machine (cont.)

depart_dn =

ANY *I*

WHEN

I \notin *moving*

dir(I) = *dn*

attracted_dn(I \mapsto *floor* \mapsto *in* \mapsto *out*) = TRUE

THEN

moving := *moving* \cup {*I*}

floor(I) := *floor(I)* - 1

END

...

Lift system: machine (cont.)

change_up_to_dn =

ANY *I*

WHEN

I \notin moving

dir(I) = up

attracted_up(I \mapsto floor \mapsto in \mapsto out) = FALSE

attracted_dn(I \mapsto floor \mapsto in \mapsto out) = TRUE

THEN

in := *in* \ {*floor(I)* \mapsto *dn*}

dir(I) := *dn*

END

...

Lift system: machine (cont.)

change_dn_to_up =

ANY *I*

WHEN

I \notin *moving*

dir(I) = *dn*

attracted_dn(*I* \mapsto *floor* \mapsto *in* \mapsto *out*) = *FALSE*

attracted_up(*I* \mapsto *floor* \mapsto *in* \mapsto *out*) = *TRUE*

THEN

in := *in* \ {*floor(I)* \mapsto *up*}

dir(I) := *up*

END

...

Lift system: notes

- Sometimes we need to check whether a lift can skip a floor while moving further to its direction or it needs to stop
- These notions (*can_continue_up* and *can_continue_down*) rely on the introduced notions *attracted_up* and *attracted_down*
- We can formalise *can_continue_up* as follows

$$I \mapsto \text{floor}(I) \notin \text{out} \wedge \text{floor}(I) \mapsto \text{dir}(I) \notin \text{in} \wedge \text{attracted_up}(\dots)$$

- Similarly, *can_continue_down* is defined as follows

$$I \mapsto \text{floor}(I) \notin \text{out} \wedge \text{floor}(I) \mapsto \text{dir}(I) \notin \text{in} \wedge \text{attracted_dn}(\dots)$$

Deconstruction a property

- $I \mapsto floor(I) \notin out$
there no people that want to get out on this floor
- $floor(I) \mapsto dir(I) \notin in$
there no people that want to get in on this floor to move in the lift direction
- $attracted_up(...)$
there are the floors above where people requested to get in or out

Lift system: context (cont.)

Introducing the corresponding higher-order definitions in the context

$$\begin{aligned} can_continue_up \in & LIFT \times (LIFT \rightarrow FLOOR) \times (LIFT \rightarrow DIRECTION) \\ & \times (FLOOR \leftrightarrow DIRECTION) \times (LIFT \leftrightarrow FLOOR) \rightarrow \text{BOOL} \end{aligned}$$
$$\begin{aligned} \forall l, floor, dir, in, out. \ can_continue_up(l \mapsto floor \mapsto dir \mapsto in \mapsto out) = \\ & \text{bool}(l \mapsto floor(l) \notin out \wedge floor(l) \mapsto dir(l) \notin in \\ & \wedge attracted_up(l \mapsto floor \mapsto in \mapsto out) = \text{TRUE}) \end{aligned}$$
$$\begin{aligned} can_continue_dn \in & LIFT \times (LIFT \rightarrow DIRECTION) \times (LIFT \rightarrow FLOOR) \\ & \times (FLOOR \leftrightarrow DIRECTION) \times (LIFT \leftrightarrow FLOOR) \rightarrow \text{BOOL} \end{aligned}$$
$$\begin{aligned} \forall l, floor, dir, in, out. \ can_continue_dn(l \mapsto floor \mapsto dir \mapsto in \mapsto out) = \\ & \text{bool}(l \mapsto floor(l) \notin out \wedge floor(l) \mapsto dir(l) \notin in \\ & \wedge attracted_dn(l \mapsto floor \mapsto in \mapsto out) = \text{TRUE}) \end{aligned}$$

END

Lift system: machine (cont.)

```
continue_up =  
ANY I  
WHEN  
     $I \in moving$   
     $dir(I) = up$   
    can_continue_up( $I \mapsto floor \mapsto dir \mapsto in \mapsto out$ ) = TRUE  
THEN  
     $floor(I) := floor(I) + 1$   
END
```

...

Lift system: machine (cont.)

```
continue_dn =  
ANY I  
WHEN  
     $I \in moving$   
     $dir(I) = dn$   
    can_continue_dn( $I \mapsto floor \mapsto dir \mapsto in \mapsto out$ ) = TRUE  
THEN  
     $floor(I) := floor(I) - 1$   
END
```

...

Lift system: machine (cont.)

```
stop_up =  
ANY /  
WHEN  
     $I \in moving$   
     $dir(I) = up$   
     $can\_continue\_up(I \mapsto floor \mapsto dir \mapsto in \mapsto out) = FALSE$   
THEN  
     $moving := moving \setminus \{I\}$   
     $out := out \setminus \{I \mapsto floor(I)\}$   
     $in := in \setminus \{floor(I) \mapsto dir(I)\}$   
END
```

...

Lift system: machine (cont.)

```
stop_dn =  
ANY /  
WHEN  
     $I \in moving$   
     $dir(I) = dn$   
     $can\_continue\_dn(I \mapsto floor \mapsto dir \mapsto in \mapsto out) = FALSE$   
THEN  
     $moving := moving \setminus \{I\}$   
     $out := out \setminus \{I \mapsto floor(I)\}$   
     $in := in \setminus \{floor(I) \mapsto dir(I)\}$   
END
```

...

Lift system: notes

- An example of the system execution:

*depart_up → continue_up → stop_up → depart_up →
continue_up → continue_up → stop_up →
change_up_to_dn → depart_dn → continue_dn → ...*

- In between this sequence, any number of the events *request_lift_from_floor* and *request_floor_from_lift* can occur
- Overall, the developed lift models demonstrate how we can handle/distribute the system complexity, by defining necessary higher-order notions in the model context and thus simplifying modelling and verifying system dynamics (machine)

Lecture 11: Outline

- Extensions (plug-ins) of the Rodin platform
- ProB: model checker and animator for Event-B
- Examples
- Homework 4: a library system

Extensions (plug-ins) of the Rodin platform

- Various functionality extensions for the Rodin platform
- Implemented as Eclipse plug-ins, which can be downloaded and installed when needed
- Additional functionality via extra menus or menu choices (buttons), Eclipse views (subwindows) or perspectives
- Different classes of extensions: editors, modelling extensions, tools for documentation, visualisation, model checking, bridging with external languages or tools, theory and proof enhancements, code generation

Extensions (plug-ins) of the Rodin platform (cont.)

- Editors: Camille (text editor), graphical editors (usually integrated with some modelling extensions)
- Modelling extensions:
 - the *UML-B* plug-in provides a “UML-like” graphical front end for Event-B
 - the *model decomposition* plug-in allows decomposition of Event-B machines/contexts (using the shared variables and the shared events decomposition)
 - the modularisation plug-in supports modular development in Event-B (including callable operations)
 - the *Flow* plug-in allows to construct and verify use cases
 - records, team-based development, mode/fault tolerance views, refactoring, ...

Extensions (plug-ins) of the Rodin platform (cont.)

- Animation/model checking/visualisation: ProB, BMotion Studio
- Documentation: ProR (integration of natural language requirements and Event-B models)
- Theory and proof: the Theory plug-in (more advanced user-defined data structures), integration with SMT solvers, theorem provers (Isabelle for Rodin)
- Code generation: Java, JML, Dafny, SQL, C

Verification by theorem proving vs model checking

- By default, Event-B (and the Rodin platform) relies on model verification by *theorem proving*
- **Advantages:** the underlying mathematical model description (semantics) is used to prove the required properties independently of how big or complex data structures are or how many different state transitions are possible
- **Disadvantages:** provers may be unable to automatically prove more complex properties. Thus, splitting model development into additional refinement steps or employing interactive theorem proving that needs extra expertise may be required

Verification by theorem proving vs model checking (cont.)

- Alternatively (or in combination with theorem proving), the Rodin platform can be extended to support model verification by *model checking*
- Model checking is a verification technique that explores (checks) all possible system states in a brute-force manner
- It is good for quick feedback, animation, or "debugging" of a model
- Moreover, it allows checking for deadlocks or temporal model properties

Verification by theorem proving vs model checking (cont.)

- **Advantages:**

- easier to implement and apply (a potential "push-button" technology),
- supports partial verification of selected properties,
- gives a counter-example and a trace leading to a detected violation in case of verification failure,
- can be applied for verification of temporal or quantitative properties.

- **Disadvantages:**

- suffers from the state-explosion problem (typically works well up to $10^8 - 10^9$ states), so finding suitable abstractions is essential,
- difficult to reason about abstract data types (which are theoretically infinite),
- it can take time and stop because of the exhausted memory

Verification by theorem proving vs model checking (cont.)

- Model checking usually relies on a generated *reachability graph* of system states
- A model checker explores all the state traces in the graph and checks the property (e.g. invariant preservation) in reached states
- Since a model checker operates on traces (not just single states), additional reachability properties can be formulated and checked
- For instance, that eventually some specific state will be reached or occurrence of some particular states will eventually lead to specific required states

$\text{critical_fault} = \text{TRUE} \longrightarrow \text{alarm} = \text{ON} \wedge \text{shutdown_mode} = \text{TRUE}$

ProB: model checker for Event-B

- ProB – a model checker and simulator for Event-B
- Integrates a separate perspective in Eclipse, allowing to initialise and simulate a model, as well as model check its desired properties
- Simulation and model checking often go hand by hand, since a reachability graph is generated for a model
- We can analyse this graph step-by-step, choosing the available execution branches as we go in our simulation
- Or, we can check it as the whole, running all possible traces to verify a given property. The result is either confirmation of a property or its violation in a specific state for a specific trace

ProB: model checker for Event-B (cont.)

The screenshot shows the ProB tool interface with several windows open:

- Events**: A list of events: choose, cancel, pay, serve, to_ready, add_items.
- State**: The main workspace showing the state table for the LTL Counter-Example model. The table has three columns: Name, Value, and Previous value. It contains entries for CO and MO states, along with formulas for sets, invariants, axioms, and guards.
- History**: A log of actions taken: serve, pay(1), choose(CHOICES2), add_items(CHOICES2,1), add_items(CHOICES2,1). It also shows the initialization step: SETUP_CONTEXT((CHOICES2,1,((CHOICES2-1))) (uninitialised state)).
- Event Error View**: Shows no errors.
- Event-B Explorer**: A tree view of Rodin Problems, including CS_security, FileTransfer, MetaSquared, NewRobots, NokiaSecurityExample, NokiaSecurityExample_Step2, RoboticSystem v.3.3, Security, Server, Services, Sluice-3, Test, Toy_Nokia, Vending, and Vending_new.
- Bottom Status Bar**: Invariants ok, no event errors detected.

ProB: model checker for Event-B (cont.)

- Allows "debugging of a model", with counter-examples providing valuable info
- Can be quickly used to fix obvious model mistakes without involving theorem proving
- Thus theorem proving and model checking/animation can be used in combination
- Symmetry reduction and other "optimisation" techniques can be used to battle the state explosion problem

ProB: properties to be verified

- Properties that can be verified in this way include invariant properties, the absence of deadlocks (when all the events are disabled), or reachability (temporal) properties
- An invariant violation or the presence of deadlocks is demonstrated by a counter example (some specific state)
- A temporal property (written using the temporal logic notation) is checked for all traces
- Animation allows us to confirm our intuition about the system behaviour, while the checked properties verify its correctness

An example for model checking: a vending machine revisited

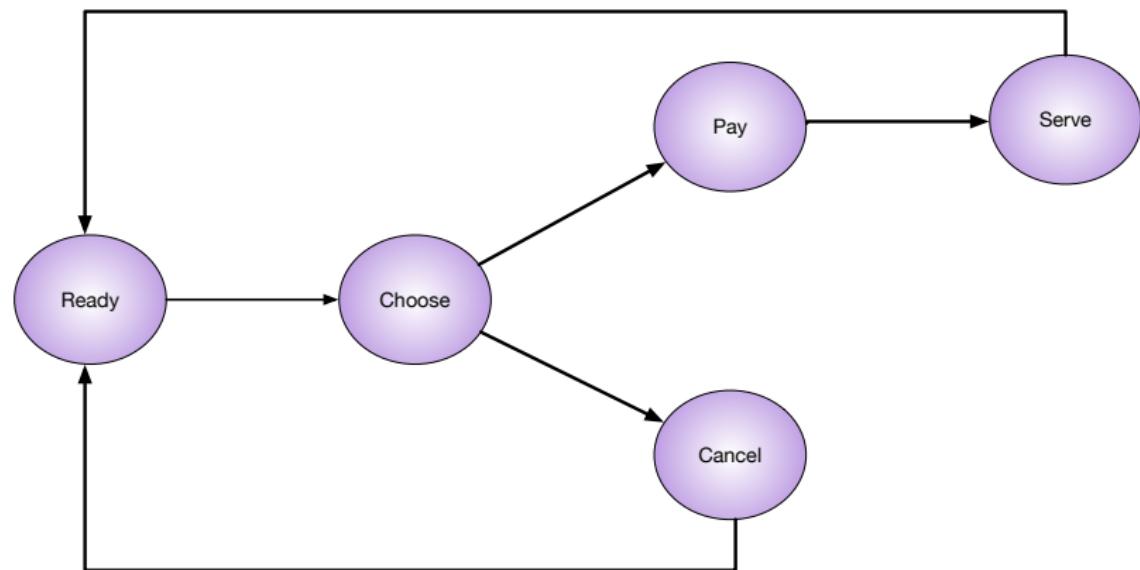
- ① The vending machine serves the customer a product from a number of available choices;
- ② The customer may select one of the available product choices;
- ③ After selection, the customer may proceed by paying for the selected product or cancelling the selection;
- ④ After sufficient payment, the selected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

Example: a vending machine (cont.)

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

Example: a vending machine (cont.)

An expected system usecase:



ProB example: notes

- Some patterns of LTL formulas:
 - $G(<\text{formula}>)$ – a formula holds all the time
 - $F(<\text{formula}>)$ – a formula holds eventually
 - $e(<\text{event1}, \dots>)$ – events enabled
 - $\{\langle\text{condition}\rangle\}$ – condition is true
- Example:
$$G (\{\text{choice} \neq \text{NONE}\} \Rightarrow F e(\text{choose}))$$

Always, after the choice is made, eventually the choosing event will become enabled again

Homework 4: a library system (requirements)

- ① A library system manages books and book readers in a library
- ② Books can be loaned to the readers of the library
- ③ There could be several copies of the same book in the library
- ④ The system should record the library books, their availability to the readers, as well as which books are currently loaned to which readers
- ⑤ There is an upper bound of the number of books (a predefined constant) that a single reader can loan
- ⑥ The last copy of a book cannot be loaned

The library system requirements (cont.)

- ⑦ The same reader cannot loan more than one copy of the same book
- ⑧ If a book is not available, a reader can be put on the waiting list for that book (this is only possible for the books with more than one copy)
- ⑨ The system must allow to loan a book (if possible), return a book, put a reader on the waiting list, add more copies of a new or already existing book
- ⑩ If the waiting list for a particular book is not empty, then the book can be loaned only for the first reader from the waiting list

Lecture 12: Outline

- Static verification vs dynamic verification (of program code)
- A framework for static verification
- The Spec# annotation language and Boogie verification engine
- Other static verification languages/engines

Reminder: the use of formal models

- Requirements formalisation and "debugging"
- Code generation (after making models sufficiently detailed by, e.g., refinement steps)
- Model-based testing (generating test suites out of system models)
- Runtime (dynamic) or static verification of annotated program code.
Model pieces are here incorporated into program code as special instructions/annotations

Static Verification vs Dynamic (Runtime) Verification

- Dynamic Verification: testing that a certain assertion/condition holds at a particular point during runtime execution
- Precompiling embedded assertions

```
assert x.length > 0; ...code...
```

into, for example,

```
if not (x.length > 0) then
```

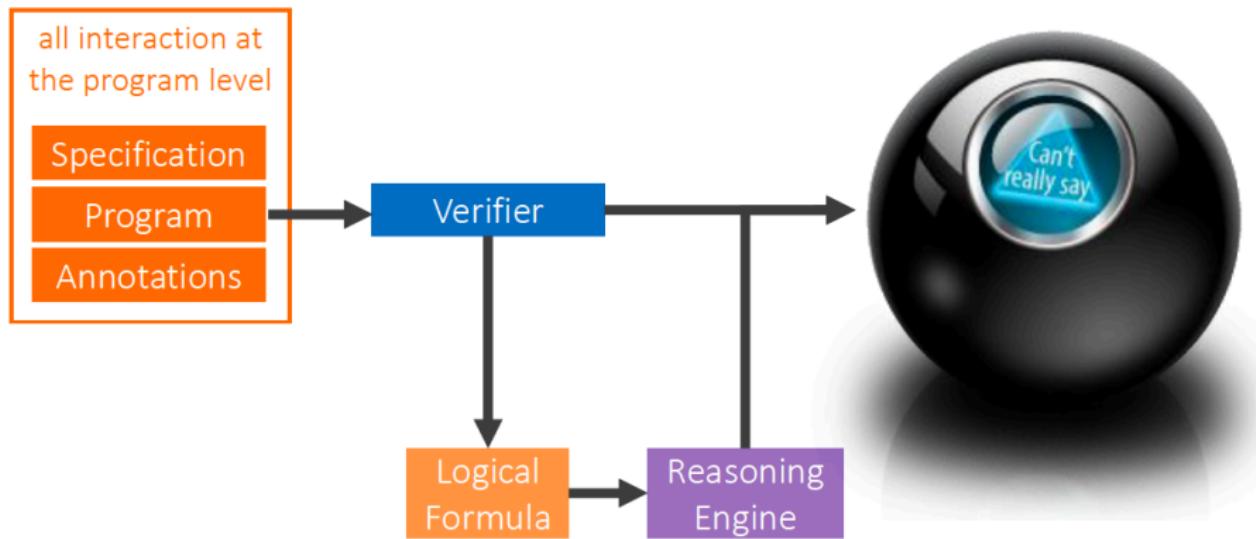
```
raise AssertionException("Assertion condition ... violated!");
```

```
...code...
```

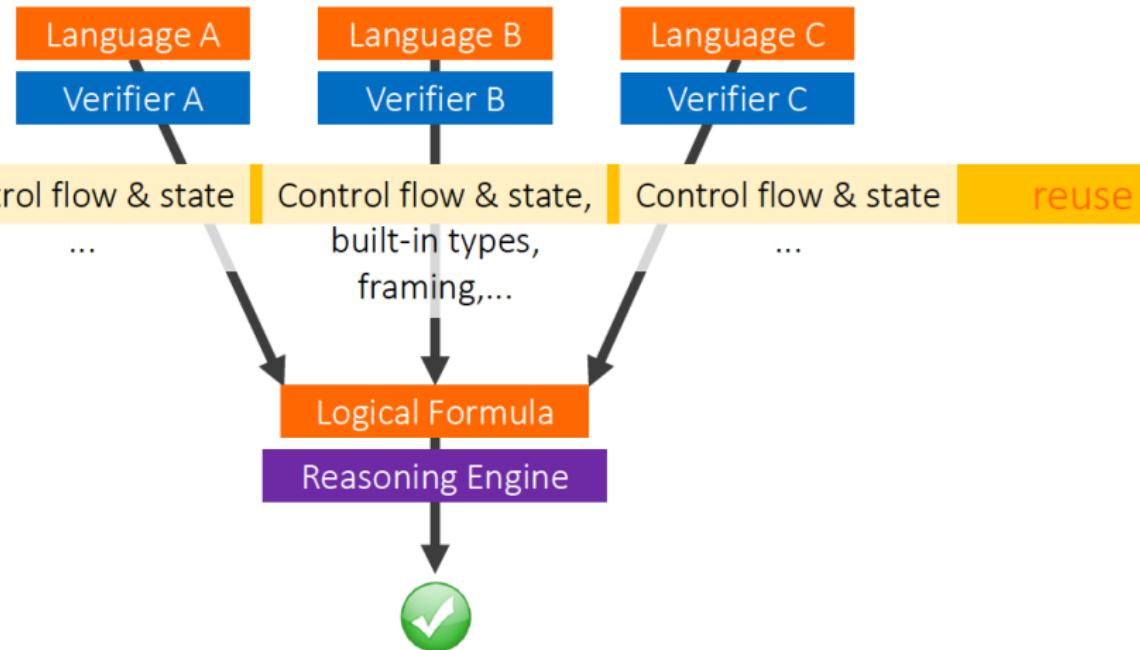
Static verification vs dynamic verification

- Static Verification: Analysing code together with embedded annotations during (pre)compilation time
- Static verification steps:
 - Precompiling of annotations together with code,
 - Generating intermediate formulas / verification conditions to verify,
 - Employing a verification engine to check conditions and get answers,
 - Incorporating these answers as precompilation results (error messages/ warnings)
- Examples: JML+Esc/Java, Sparc-Ada, Spec#, code contracts, Sing#, Eiffel

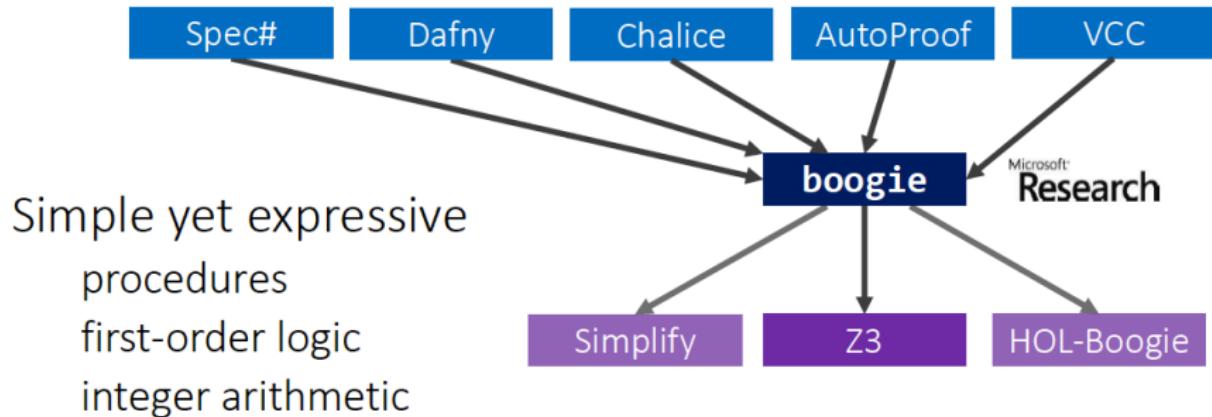
"Auto-active" verification



Verifying imperative programs



The Boogie intermediate verifier/verification language



How do we use Spec#?

- The programmer writes each class containing methods and their specification together in a Spec# source file (similar to Eiffel, similar to Java + JML)
- Invariants that constrain the data fields of objects may also be included
- We then run the verifier
- The verifier is run like the compiler—either from the IDE or the command line.
 - In either case, this involves just pushing a button, waiting, and then getting a list of compilation/verification error messages, if they exist.
 - Interaction with the verifier is done by modifying the source file.

Spec# at Microsoft

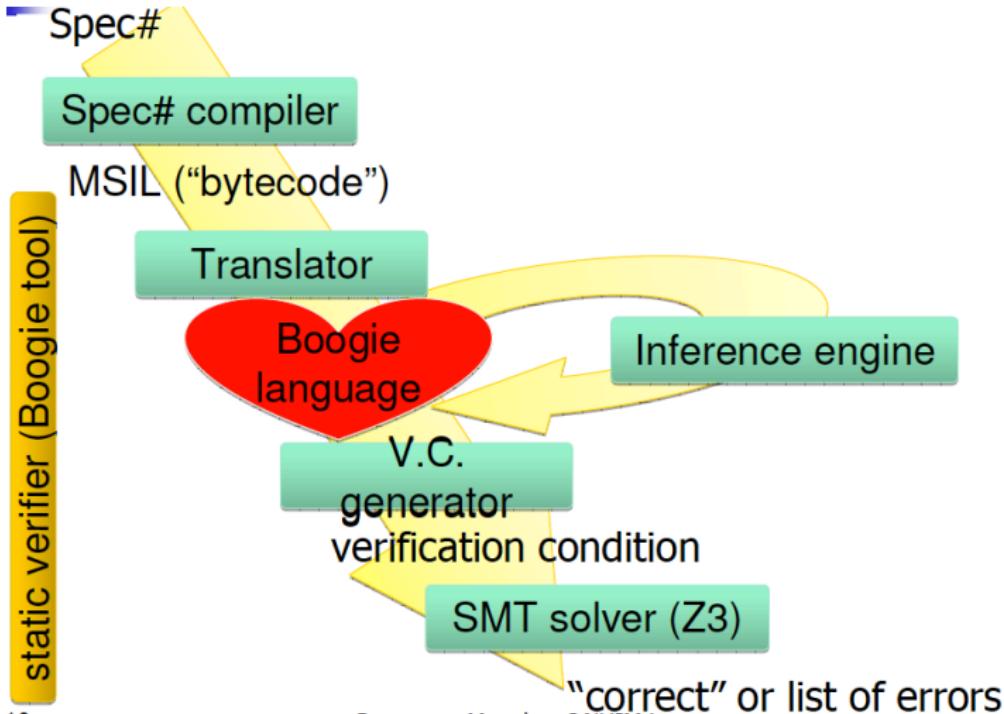
- <http://research.microsoft.com/en-us/projects/specsharp/>
- Spec# is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# with constructs for non-null types, preconditions, postconditions, and object invariants.
- Spec# is an extension of the object-oriented language C#. It extends the type system to include non-null types and checked exceptions. It provides method contracts in the form of pre- and postconditions as well as object invariants.

Spec# at Microsoft

- The Spec# compiler. Integrated into the Microsoft Visual Studio development environment for the .NET platform. Recently, incorporated as a part of the Code Contracts library;
- The compiler statically enforces non-null types, emits run-time checks for method contracts and invariants, and records the contracts as metadata for consumption by downstream tools;
- The Spec# static program verifier – Boogie. Generates logical verification conditions from a Spec# program. Internally, it uses an automatic theorem prover that analyses the verification conditions to prove the correctness of the program or find errors in it.

- Static verification checks all executions
- Spec# characteristics
 - Sound modular verification
 - Focus on automation of verification rather than full functional correctness of specifications
 - No termination verification
 - No verification of temporal properties
 - No arithmetic overflow checks

Spec# verifier architecture



- Z3 is a high-performance theorem prover being developed at Microsoft Research
- Z3 supports linear real and integer arithmetic, fixed-size bit-vectors, extensional arrays, uninterpreted functions, and quantifiers
- Z3 is integrated with a number of program analysis, testing, and verification tools from Microsoft Research. These include: VCC, Spec#, Boogie, Pex, Yogi, Vigilante, SLAM, F7, F*, SAGE, VS3, FORMULA, and HAVOC

"Hello World" program in Spec#?

```
using System;
using Microsoft.Contracts;
public class Program
{
    public static void Main(string![]! args)
    {
        Console.WriteLine("Spec# says hello!");
        Console.Read();
    }
}
```

Non-Null Types

- Many errors in modern programs manifest themselves as null-dereference errors
- Spec# tries to eradicate all null dereference errors
- In C#, each reference type T includes the value `null`
- In Spec#, type T! contains only references to objects of type T (not `null`)

```
int []! xs;
```

declares an array called xs which cannot be `null`

Non-Null Types (cont.)

- If you decide that it's the caller's responsibility to make sure the argument is not null, Spec# allows you to record this decision concisely using an exclamation point
- Spec# will also enforce the decision at call sites returning **Error: null is not a valid argument** if a null value is passed to a method that requires a non null parameter

Non-Null Example

```
using System;
using Microsoft.Contracts;
class NonNull
{
    public static void Clear(int[] xs)
    {
        for (int i = 0; i < xs.Length; i++)
        {
            xs[i] = 0;
        }
    }
}
```

Where is the *possible null dereference?*

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
    public static void Clear(int[] xs)
    {
        for (int i = 0; i < xs.Length; i++) //Warning: Possible null dereference?
        {
            xs[i] = 0; //Warning: Possible null dereference?
        }
    }
}
```

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
    public static void Clear(int[] ! xs)
    {
        for (int i = 0; i < xs.Length; i++) // No Warning due to !
        {
            xs[i] = 0; // No Warning due to !
        }
    }
}
```

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
    public static void Clear(int[] ! xs)
    {
        for (int i = 0; i < xs.Length; i++)
        {
            xs[i] = 0;
        }
    }
}
```

```
class ClientCode
{
    static void Main()
    {
        int[] xs = null;
        NonNull.Clear(xs);
    }
}
```

Non-Null Example (cont.)

```
using System;
using Microsoft.Contracts;
class NonNull
{
    public static void Clear(int[] ! xs)
    {
        for (int i = 0; i < xs.Length; i++)
        {
            xs[i] = 0;
        }
    }
}
```

“Null cannot be used where a non-null value is expected”

```
class ClientCode
{
    static void Main()
    {
        int[] xs = null;
        NonNull.Clear(xs);
    }
}
```

Difference: assertions and assumptions!

Assert Statements

```
public class Assert
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            if (arg.StartsWith("Hello"))
            {
                assert 5 <= arg.Length; // runtime check
                char ch = arg[2];
                Console.WriteLine(ch);
            }
        }
    }
}
```

<Assert.ssc>

Assert Statements (cont.)

```
public class Assert
{
    public static void Main(string[] args)
    {
        foreach (string arg in args)
        {
            if (arg.StartsWith("Hello"))
            {
                assert 5 < arg.Length; // runtime error
                char ch = arg[2];
                Console.WriteLine(ch);
            }
        }
    }
}
```

Design by Contract

- Every public method has a precondition and a postcondition
- The **precondition** expresses the constraints under which the method will function properly
- The **postcondition** expresses what will happen when a method executes properly
- Pre- and postconditions are checked
- Preconditions and postconditions are **side-effect free** boolean-valued expressions - i.e. they evaluate to true/false and can't use ++

The Swap Contract

```
static void Swap(int[] a, int i, int j)
requires
modifies
ensures
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

The Swap Contract(cont.)

```
static void Swap(int[]! a, int i, int j)
    requires 0 <= i && i < a.Length;
    requires 0 <= j && j < a.Length;
    modifies a[i], a[j];
    ensures a[i] == old(a[j]);
    ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

The Swap Contract (cont.)

```
static void Swap(int[]! a, int i, int j)
    requires 0 <= i && i < a.Length;
    requires 0 <= j && j < a.Length;
    modifies a[i], a[j];
    ensures a[i] == old(a[j]);
    ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

requires annotations
denote preconditions

Requires Clause

```
static void Swap(int[]! a, int i, int j)
    requires 0 <= i && i < a.Length;
    requires 0 <= j && j < a.Length;
    modifies a[i], a[j];
    ensures a[i] == old(a[j]);
    ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

frame conditions limit
the parts of the program state
that the method is allowed to modify.

Referring to Old Values

```
static void Swap(int[]! a, int i, int j)
    requires 0 <= i && i < a.Length;
    requires 0 <= j && j < a.Length;
    modifies a[i], a[j];
    ensures a[i] == old(a[j]);
    ensures a[j] == old(a[i]);
{
    int temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

old(a[j]) denotes the value of *a[j]* on entry to the method

Referring to the Result

```
static int F( int p )
ensures 100 < p ==> result == p - 10;
ensures p <= 100 ==> result == 91;
{
    if ( 100 < p )
        return p - 10;
    else
        return F( F(p+11) );
}
```

result denotes the value returned by the method

Spec# Constructs so far

- `==>` short-circuiting implication
- `<==>` if and only if
- **result** denotes method return value
- **old(E)** denotes E evaluated in method's pre-state
- **requires** E; declares precondition
- **ensures** E; declares postcondition
- **modifies** w; declares what a method is allowed to modify
- **assert** E; in-line assertion

- **modifies** w where w is a list of:
 - p.x field x of p
 - p.* all fields of p
 - p.** all fields of all peers of p
 - **this.*** default modifies clause, if **this**-dot-something is not mentioned in modifies clause
 - **this.0** disables the "**this.***" default
 - a[i] element i of array a
 - a[*] all elements of array a

Modifies Clauses (cont.)

- We can use a postcondition to exclude some modifications (from the default `this.*`)
- We can use a **modifies** clause to allow certain modifications
- `x++, x--` in a method \Rightarrow must have a **modifies** clause

Loop Invariants

- Statically verifying (calculating whether some condition is true at a certain point) is relatively easy for assignments, if statements, calls etc.
- The tough part – calculating what is true after a loop that may involve a significant number of iterating statements
- The problem can be often solved by submitting loop invariants – hints what is supposed to be true before and after each loop iteration
- Loop invariants also often formulate what intermediate results are achieved after each step

Loop Invariants: Computing Square by Addition

```
public int Square(int n)
    requires 0 <= n;
    ensures result == n*n;
{
    int r = 0;
    int x = 1;
    for (int i = 0; i < n; i++)
        invariant i <= n;
        invariant r == i*i;
        invariant x == 2*i + 1;
    {
        r = r + x;
        x = x + 2;
    }
    return r;
}
```

Square(3)

- $r = 0$ and $x = 1$ and $i = 0$
- $r = 1$ and $x = 3$ and $i = 1$
- $r = 4$ and $x = 5$ and $i = 2$
- $r = 9$ and $x = 7$ and $i = 3$

Loop Invariants (cont.)

- The pre-compiler makes the loop invariants into assertions to be checked
- Moreover, the verifier uses the invariant information to verify postconditions (**ensures** or **assert** statements occurring after the loop body)
- Formally:

$$\textit{loop_invariants} \wedge \neg \textit{loop_condition} \Rightarrow \textit{loop_postcondition}$$

Loop Invariants: Integer Square Root

```
public static int ISqrt(int x)
    requires 0 <= x;
    ensures result*result <= x && x < (result+1)*(result+1);
{
    int r = 0;
    while ((r+1)*(r+1) <= x)
        invariant r*r <= x;
    {
        r++;
    }
    return r;
} <ISqrt.ssc>
```

Loop Invariants: Integer Square Root (cont.)

```
public static int ISqrt1(int x)
    requires 0 <= x;
    ensures result*result <= x && x < (result+1)*(result+1);
{
    int r = 0; int s = 1;
    while (s<=x)
        invariant r*r <= x;
        invariant s == (r+1)*(r+1);
    {
        r++;
        s = (r+1)*(r+1);
    }
    return r;
}
```

Examples:

- **forall** {int k **in** (0: a.Length); a[k] > 0};
- **exists** {int k **in** (0: a.Length); a[k] > 0};
- **exists unique** {int k **in** (0: a.Length); a[k] > 0};

```
void Square(int[]! a)
  modifies a[*];
  ensures forall{int i in (0: a.Length); a[i] == i*i};
```

Loop Invariants (cont.)

```
void Square(int[]! a)
  modifies a[*];
  ensures forall{int i in (0: a.Length); a[i] == i*i};
{
  int x = 0; int y = 1;
  for (int n = 0; n < a.Length; n++)
    invariant 0 <= n && n <= a.Length;
    invariant forall{int i in (0: n); a[i] == i*i};
  {
    a[n] = x;
    x += y;
    y += 2;
  }
}
```

<SqArray.ssc>

Error Message from Boogie

Spec# program verifier version 2, Copyright (c) 2003- 2010, Microsoft.

Error: After loop iteration: Loop invariant might not hold: forall{int i in (0: n); a[i] == i*i}

Spec# program verifier finished with 1 verified, 1 error

Loop Invariants (cont.)

```
void Square(int[]! a)
    modifies a[*];
    ensures forall{int i in (0: a.Length); a[i] == i*i};
{
    int x = 0; int y = 1;
    for (int n = 0; n < a.Length; n++)
        invariant 0 <= n && n <= a.Length;
        invariant forall{int i in (0: n); a[i] == i*i};
        invariant x == n*n && y == 2*n + 1;
    {
        a[n] = x;
        x += y;
        y += 2;
    }
}
```

Inferred by /infer:
Inferred by default

Lecture 13: Outline

- Static verification (part 2)
- Model checking of temporal and timing properties
- The Uppaal model checker

Some Spec# constructs

- `==>` short-circuiting implication
- `<==>` if and only if
- **result** denotes method return value
- **old(E)** denotes E evaluated in method's pre-state
- **requires** E; declares precondition
- **ensures** E; declares postcondition
- **modifies** w; declares what a method is allowed to modify
- **assert** E; in-line assertion

Examples:

- **forall** {int k **in** (0: a.Length); a[k] > 0};
- **exists** {int k **in** (0: a.Length); a[k] > 0};
- **exists unique** {int k **in** (0: a.Length); a[k] > 0};

```
void Square(int[]! a)
  modifies a[*];
  ensures forall{int i in (0: a.Length); a[i] == i*i};
```

Examples:

- **sum** {int k **in** (0: a.Length); a[k]};
- **product** {int k **in** (1..n); k};
- **min** {int k **in** (0: a.Length); a[k]};
- **max** {int k **in** (0: a.Length); a[k]};
- **count** {int k **in** (0: n); a[k] % 2 == 0};

Intervals:

- The half-open interval {int i **in** (0: n)}
means i satisfies $0 \leq i < n$
- The closed (inclusive) interval {int k **in** (0..n)}
means i satisfies $0 \leq i \leq n$

Comprehensions in Spec# (cont.)

We may also use **filters**:

- `sum {int k in (0: a.Length), 5<=k; a[k]};`
- `product {int k in (0..100), k % 2 == 0; k};`

Note that the following two expressions are equivalent:

- `sum {int k in (0: a.Length), 5<=k; a[k]};`
- `sum {int k in (5: a.Length); a[k]};`

Loop Invariants (cont.)

```
public static int SumEvens(int[]! a)
ensures result == sum{int i in (0: a.Length) | a[i] % 2 == 0; a[i]}
{
    int s = 0;
    for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length;
        invariant s == sum{int i in (0:n), a[i] % 2 == 0; a[i]};

    {
        if (a[n] % 2 == 0)
        {
            s += a[n];
        }
    }
    return s;
}
```

Filters the even values
From the quantified range

Invariant variations: Sum0

```
public static int Sum0(int[]! a)
ensures result == sum{int i in (0 : a.Length); a[i ]};
{
    int s = 0;
    for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length && s == sum{int i in (0: n); a[i]};
    {
        s += a[n];
    }
    return s;
}
```

This loop invariant focuses on what has been summed so far.

Invariant variations: Sum1

```
public static int Sum1(int[] a)
ensures result == sum{int i in (0 : a.Length); a[i]};
{  int s = 0;
   for (int n = 0; n < a.Length; n++)
      invariant n <= a.Length &&
         s + sum{int i in (n: a.Length); a[i]}
                     == sum{int i in (0: a.Length); a[i]}
   {
      s += a[n];
   }
   return s;
}
```

This loop invariant focuses on what is yet to be summed.

The *count* Quantifier

```
public int Counting(int[]! a)
ensures result == count{int i in (0: a.Length); a[i] == 0};

{
    int s = 0;
    for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length;
        invariant s == count{int i in (0: n); a[i] == 0};

    {
        if (a[n]== 0) s = s + 1;
    }
    return s;
}
```

Counts the number of
0's in an int []! a;

The *min* Quantifier

```
public int Minimum()
    ensures result == min{int i in (0: a.Length); a[i]};
{
    int m = System.Int32.MaxValue;
    for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length;
        invariant m == min{int i in (0: n); a[i]};
    {
        if (a[n] < m)
            m = a[n];    Calculates the minimum value
    }                                in an int []! a;
}
return m;
}
```

The *max* Quantifier

```
public int MaxEven()
ensures result == max{int i in (0: a.Length), a[i] % 2== 0;a[i]};
{
    int m = System.Int32.MinValue;
    for (int n = 0; n < a.Length; n++)
        invariant n <= a.Length;
        invariant m == max{int i in (0: n), a[i] % 2== 0; a[i]};
    {
        if (a[n] % 2== 0 && a[n] > m)
            m = a[n];    Calculates the maximum even
                           value in an int []! a;
    }
    return m;
}
```

How to help the verifier ...

Recommendations when using comprehensions:

- Write specifications in a form that is as close to the code as possible.
- When writing loop invariants, write them in a form that is as close as possible to the postcondition

In our *SegSum* example where we summed the array elements $a[i] \dots a[j-1]$, we could have written the postcondition in either of two forms:

```
ensures result == sum{int k in (i: j); a[k]};  
ensures result ==  
    sum{int k in (0: a.Length), i <= k && k < j; a[k]};
```

How to help the verifier ... (cont.)

Recommendation: When writing loop invariants, write them in a form that is as close as possible to the postcondition.

```
ensures result == sum{int k in (i: j); a[k]};  
invariant i <= n && n <= j;
```

```
invariant s == sum{int k in (i: n); a[k]};
```

OR

```
ensures result ==  
    sum{int k in (0: a.Length), i <= k && k < j; a[k]};  
invariant 0 <= n && n <= a.Length;  
invariant s == sum{int k in (0: n), i <= k && k < j; a[k]};
```

- Specifying the rules for using methods is achieved through contracts, which spell out what is expected of the caller (**preconditions**) and what the caller can expect in return from the implementation (**postconditions**).
- To specify the design of an implementation, we use an assertion involving the data in the class called an **object invariant**.
- Each objects data fields must satisfy the invariant at all **stable** times

Object invariant example

```
public class RockBand
{
    int shows;
    int ads;

    invariant shows <= ads;
    public void Play()
    {
        ads++;
        shows++;
    }
}
```

<RockBand1.ssc>

Broken object invariant example

```
public class RockBand
{
    int shows;
    int ads;

    invariant shows <= ads;
    public void Play()
    {
        shows++;
        ads++;
    }
}
```

Broken object invariant example (cont.)

```
public class RockBand
```

```
    public void addShow() {  
        shows++;  
        ads++;  
    }  
}
```

RockBand2.ssc(13,5): Error: Assignment to field
RockBand.shows of non-exposed target object may
break invariant: shows <= ads

Spec# program verifier finished with 4 verified, 1 error

```
    public void addShow() {  
        shows++;  
        ads++;  
    }  
}
```

Fixed object invariant example

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
    {
        expose(this)
        {
            shows++;
            ads++;
        }
    }
}
```

Method Reentrancy

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
    {
        expose(this)
        {
            shows++;
            Play();
            ads++;
        }
    }
}
```

Method Reentrancy (cont.)

Verifying RockBand.Play ...

RockBand4.ssc(20,3): Error:

The call to RockBand.Play()
requires target object to be peer consistent

```
    } // end of exposed(ads)
    {
        shows++;
        Play();
        ads++;
    }
}
```

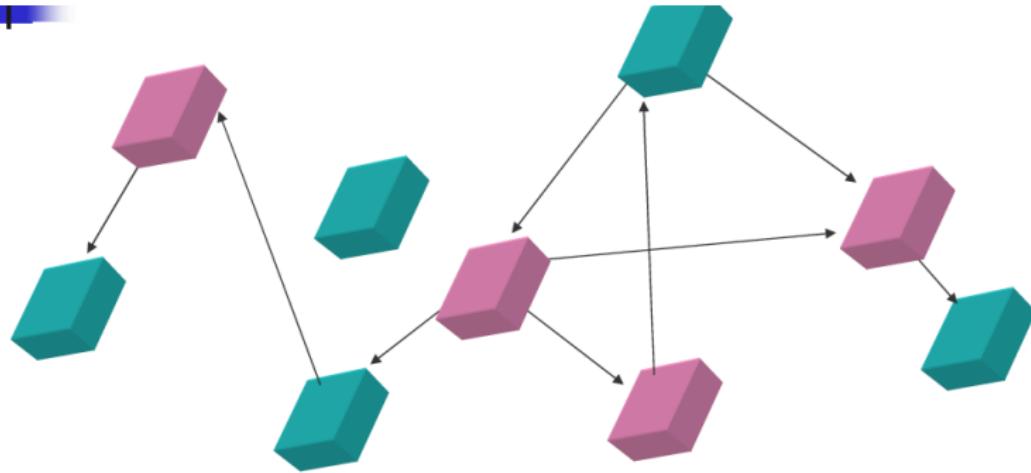
Method Reentrancy (cont.)

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
    {
        expose(this)
        {
            shows++;
            ads++;
        }
        Play();
    }
}
```

<RockBand6.ssc>

- **Mutable**
 - Object invariant might be violated
 - Field updates are allowed
- **Valid**
 - Object invariant holds
 - Field updates allowed only if they maintain the invariant

Object States (cont.)



To Mutable and Back

```
public class RockBand
{
    int shows;
    int ads;
    invariant shows <= ads;
    public void Play()
        modifies shows, ads;
        ensures ads==old(ads)+1 && shows ==old(shows)+1
    {
        expose(this) {
            shows++;
            ads++;
        }
    }
}
```

changes this from valid to mutable

can update ads and shows because this.mutable

changes this from mutable to valid

To Mutable and Back

```
class Counter{  
    int c;  
    bool even;  
    invariant 0 <= c;  
    invariant even <==> c % 2 == 0;  
    ...  
    public void Inc ()  
        modifies c;  
        ensures c == old(c)+1:  
    {   expose(this) {  
        c++;  
        even = !even ;  
    }  
}
```

changes this
from valid to mutable

can update c and even,
because this.mutable

changes this
from mutable to valid

Object Invariants: Summary

```
class Counter{  
    int c;  
    bool even;  
    invariant 0 <= c;  
    invariant even <==> c % 2 == 0;  
  
    public Counter()  
    {      c = 0;      ←  
          even = true; ←  
    } ←  
  
    public void Inc ()  
        modifies c;  
        ensures c == old(c)+1;  
    {      expose (this) {  
          c++;  
          even = !even ;  
      }      } } } }
```

The invariant may be broken in the constructor

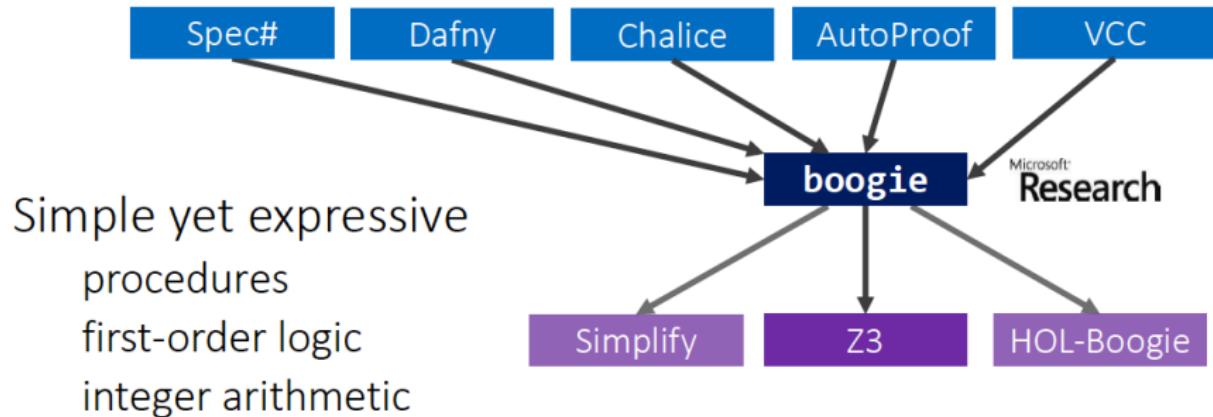
The invariant must be established & checked after construction

The object invariant may be broken within an expose block

Specification inheritance

- Spec# verifies a call to a virtual method M against the specification of M in the static type of the receiver and enforces that all overrides of M in subclasses live up to that specification
- An overriding method inherits the precondition, postcondition, and **modifies** clause from the methods it overrides.
- It may declare additional postconditions, but not additional preconditions or **modifies** clauses because a stronger precondition or a more permissive **modifies** clause would come as a surprise to a caller of the superclass method

The Boogie Intermediate Verifier/Verification Language



Some Research Languages that use Boogie as an Intermediate Language

- **Chalice**: specification and verification of concurrent programs using shared memory and mutual exclusion via locks. Chalice supports dynamic object creation, dynamic thread creation (fork and join), mutual-exclusion and readers-writers locks, monitor invariants, thread pre- and postconditions
- **Dafny**: an object-based language where specifications are written in the style of dynamic frames. The Dafny programming language is designed to support the static verification of programs. It is imperative, sequential, supports generic classes, dynamic allocation, and inductive datatypes, and builds in specification constructs

Some Research Languages that use Boogie as an Intermediate Language

- **AutoProof**, a verification tool that translates Eiffel programs to Boogie and uses the Boogie verifier to prove them. AutoProof fully supports several advanced object-oriented features including polymorphism, inheritance, and function objects
- **VCC** is a tool that proves correctness of annotated concurrent C programs or finds problems in them. VCC extends C with design by contract features, like pre- and postcondition as well as type invariants
- **HAVOC** is a tool for specifying and checking properties of systems software written in C, in the presence of pointer manipulations, unsafe casts and dynamic memory allocation. The assertion logic of HAVOC allows the expression of properties of linked lists and arrays

Microsoft Code Contracts

- Strongly influenced by Spec#;
Focus on runtime verification and testing
- **Code Contracts** provide a language-agnostic way to express coding assumptions in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants
- Contracts act as checked documentation of your external and internal APIs. The contracts are used to improve testing via runtime checking, enable static contract verification, and documentation generation
- Improved testability: each contract acts as an oracle, giving a test run a pass/fail indication; automatic testing tools can take advantage of contracts to generate more meaningful unit tests

Verification by model checking (reminder)

- Model checking usually relies on a generated *reachability graph* of system states
- A model checker explores all the state traces in the graph and checks the property (e.g. invariant preservation) in reached states
- Since a model checker operates on traces (not just single states), additional reachability properties can be formulated and checked
- For instance, that eventually some specific state will be reached or occurrence of some particular states will eventually lead to specific required states

$critical_fault = \text{TRUE} \longrightarrow alarm = \text{ON} \wedge shutdown_mode = \text{TRUE}$

Model checking of temporal and timing system properties

- One of the advantages of model checking is a possibility to check eventual reachability of certain desirable states
- Such properties are usually called liveness (in contrast, invariant properties are often called safety ones)
- Often, we need to enforce a time constraint on such eventual reachability
- Example, "if critical fault occurs, then, within the required time limit T , a desired safe state should be reached by the system"
- To reason about such properties, we need to introduce real time into our system models

The Uppaal tool

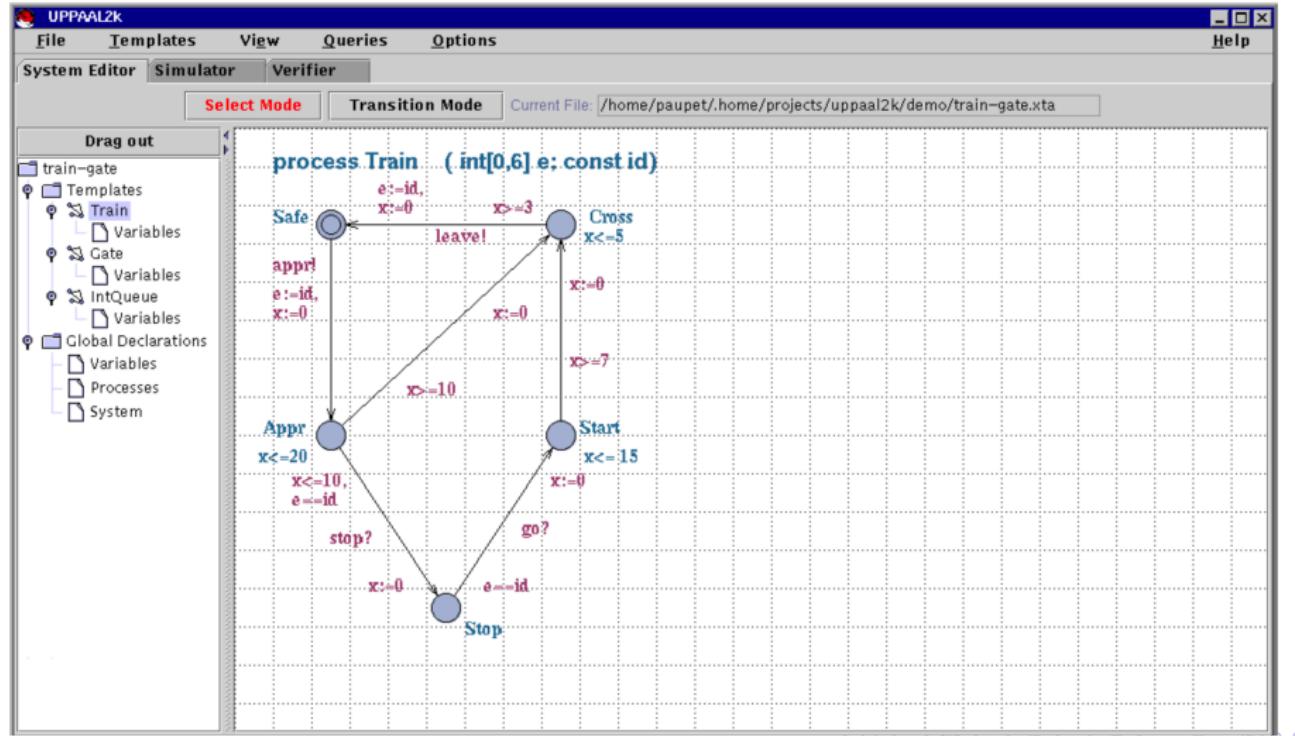
- Uppaal – integrated tool environment for modelling, simulation and verification of real-time systems
- Developed jointly by Uppsala (Sweden) and Aalborg (Denmark) universities
- It is appropriate for the systems that can be modelled as a collection of non-deterministic processes (timed automata) with finite control structure and real-valued clocks, communicating through channels or shared variables
- Typical application areas: real-time controllers, communication protocols, etc.

The Uppaal tool (cont.)

- Uppaal consists of 3 main parts: a description language, a simulator, and a model-checker
- The Uppaal language allows us to describe (model) the system behaviour as networks of automata extended with clocks and data variables. Creating of such models is facilitated by a graphical system editor
- The simulator is a validation tool which enables examination of possible dynamic system executions
- The model checker does exhaustive checking of the dynamic system behaviour and allows to verify invariant and liveness (reachability) properties of a system

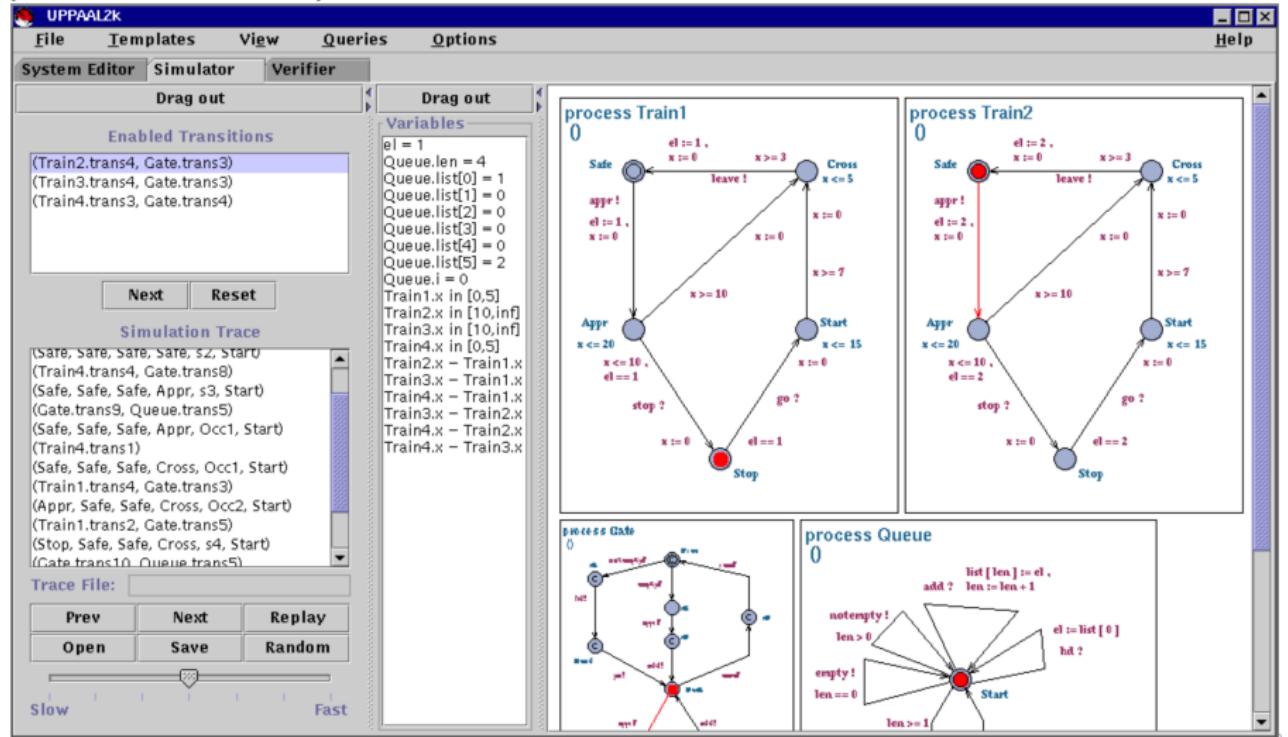
The Uppaal graphical editor

Allows modelling a system process as an automata. The circles and arrows represent system states and transitions. Time progress – within in a state



The Uppaal simulator

Provides visualisation of possible dynamic behaviours. Also shows traces (counter-examples) generated by the model checker



The Uppaal model checker

Allows to formulate and verify invariant/safety and reachability/liveness system properties by exploring the system state space

The screenshot shows the UPPAAL2K interface with the following components:

- Menu Bar:** File, Templates, View, Queries, Options, Help.
- Toolbar:** System Editor, Simulator, Verifier.
- Overview Panel:** Displays a list of properties (P0 to P7) with their formulas and status indicators (green, red, grey).
 - P0: $E \neq \text{Gate.Occ1}$ (Green)
 - P1: $E \neq \text{Train1.Cross}$ (Green)
 - P2: $E \neq \text{Train2.Cross}$ (Green)
 - P3: $E \neq (\text{Train1.Cross} \text{ and } \text{Train2.Stop})$ (Green)
 - P4: $E \neq (\text{Train1.Cross} \text{ and } \text{Train2.Stop} \text{ and } \text{Train3.Stop} \text{ and } \text{Train4.Stop})$ (Red)
 - P5: $E \neq (\text{Train1.Cross} \text{ and } \text{Train2.Stop} \text{ and } \text{Train3.Stop} \text{ and } \text{Train4.Stop})$ (Green)
 - P6: $E \neq \text{Train1.Cross}$ (Grey)
 - P7: $A[] \text{not}(\text{Train1.Cross} \text{ or } \text{Train2.Cross} \text{ or } \text{Train3.Cross} \text{ or } \text{Train4.Cross}) \backslash \text{or}(\text{Train2.Cross} \text{ and } (\text{Train3.Cross} \text{ or } \text{Train4.Cross}))$ (Green)
- Model Check Buttons:** Insert, Remove, Comments.
- Query Panel:** Displays the formula $E \neq \text{Gate.Occ1}$.
- Comment Panel:** Displays the note "Gate can receive (and store in queue) msg's from approaching trains."
- Status Panel:** Displays the status message "Property is satisfied." followed by the formula $E \neq (\text{Train1.Cross} \text{ and } \text{Train2.Stop} \text{ and } \text{Train3.Stop} \text{ and } \text{Train4.Stop})$.

Patterns of checked temporal properties

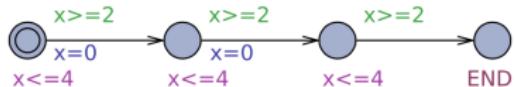
- $E<> p$: there exists an execution path where p eventually holds;
- $A[] p$: for all paths p always holds (invariant);
- $E[] p$: there is an execution path where p always holds;
- $A<> p$: for all paths p will eventually hold;
- $p \rightarrowtail q$: whenever p holds, q will eventually hold.

The Uppaal SMC tool

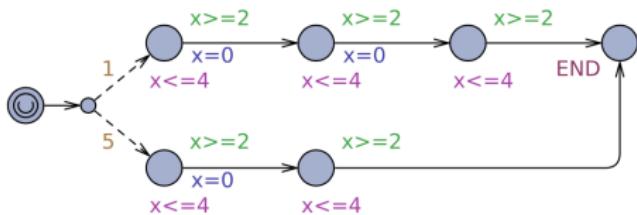
- Recently, the Uppaal tool was extended to support statistical model checking (SMC), where properties are checked with some statistical confidence (probabilities)
- The Uppaal language for modelling real-time systems has also been extended with probabilistic transition branching and clock rates (probabilistic distribution)
- The exhaustive model checking of the system state space is replaced with simulating the system behaviour for some number of "system runs" and accumulating the statistical results
- Timed automata are thus augmented to become stochastic timed automata

The Uppaal SMC tool (cont.)

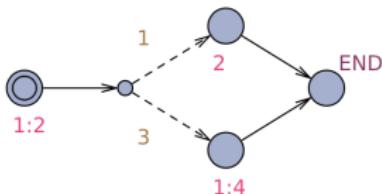
The language is extended with probabilistic transition branching (in 2nd and 3rd diagrams) and clock rates (in 3rd diagram)



(a) A_1 .

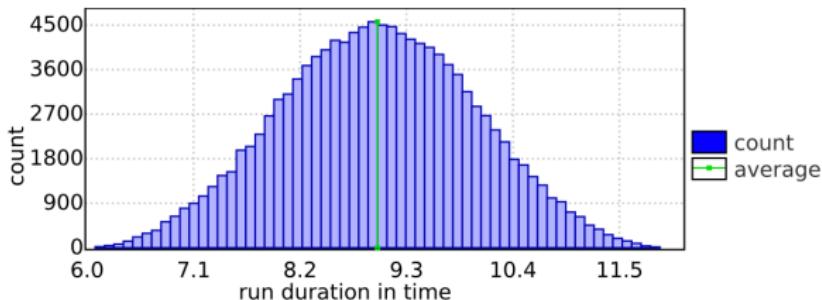


(b) A_2 .

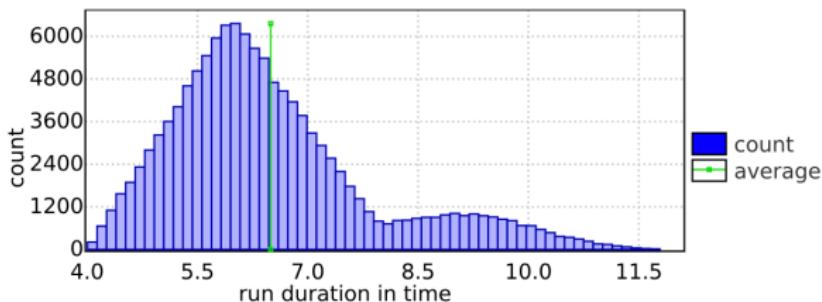


The Uppaal SMC tool (cont.)

Visualisation of distribution of reachability time



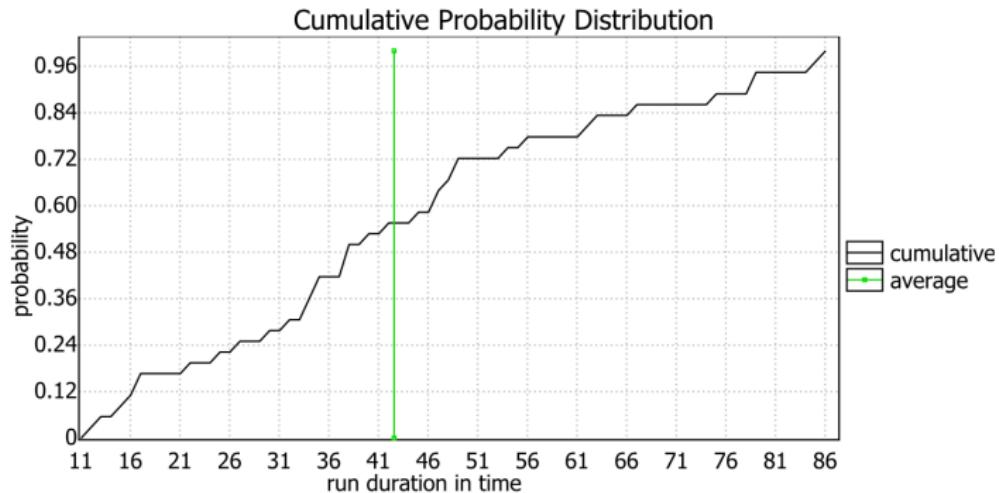
(a) A_1 arrival to END.



(b) A_2 arrival to END.

The Uppaal SMC tool (cont.)

The cumulative probability distribution of reaching a desired state within the given time



Parameters: $\alpha=0.05$, $\varepsilon=0.05$, bucket width=1, bucket count=75

Runs: 36 in total, 36 (100%) displayed, 0 (0%) remaining

Span of displayed sample: [11.0447, 85.2899]

Mean of displayed sample: 42.5735 ± 7.10607 (95% CI)

Lecture 14: Outline

- The Uppaal model checker (part 2)
- Two case studies with Uppaal

Model checking of temporal and timing system properties (reminder)

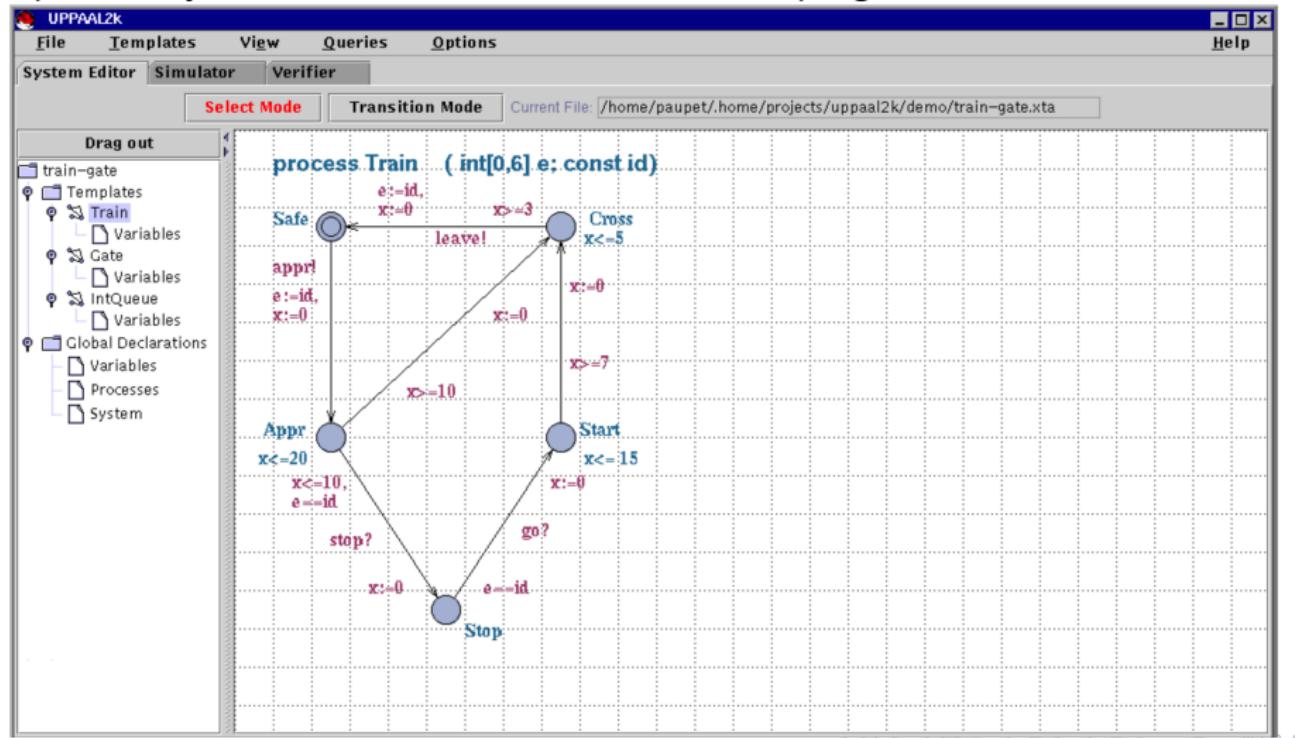
- One of the advantages of model checking is a possibility to check eventual reachability of certain desirable states
- Often, we need to enforce a time constraint on such eventual reachability. To reason about such properties, we need to introduce real time into our system models
- Statistical model checking allows us to verify more complex systems and properties, however with some statistical confidence instead of complete certainty

The Uppaal tool (reminder)

- Uppaal – integrated tool environment for modelling, simulation and verification of real-time systems
- Developed jointly by Uppsala (Sweden) and Aalborg (Denmark) universities
- Uppaal consists of 3 main parts: a description language, a simulator, and a model-checker
- Recently, the Uppaal tool was extended to support statistical model checking (SMC), where properties are checked with some statistical confidence (probabilities)
- The exhaustive model checking of the system state space is replaced with simulating the system behaviour for some number of "system runs" and accumulating the statistical results

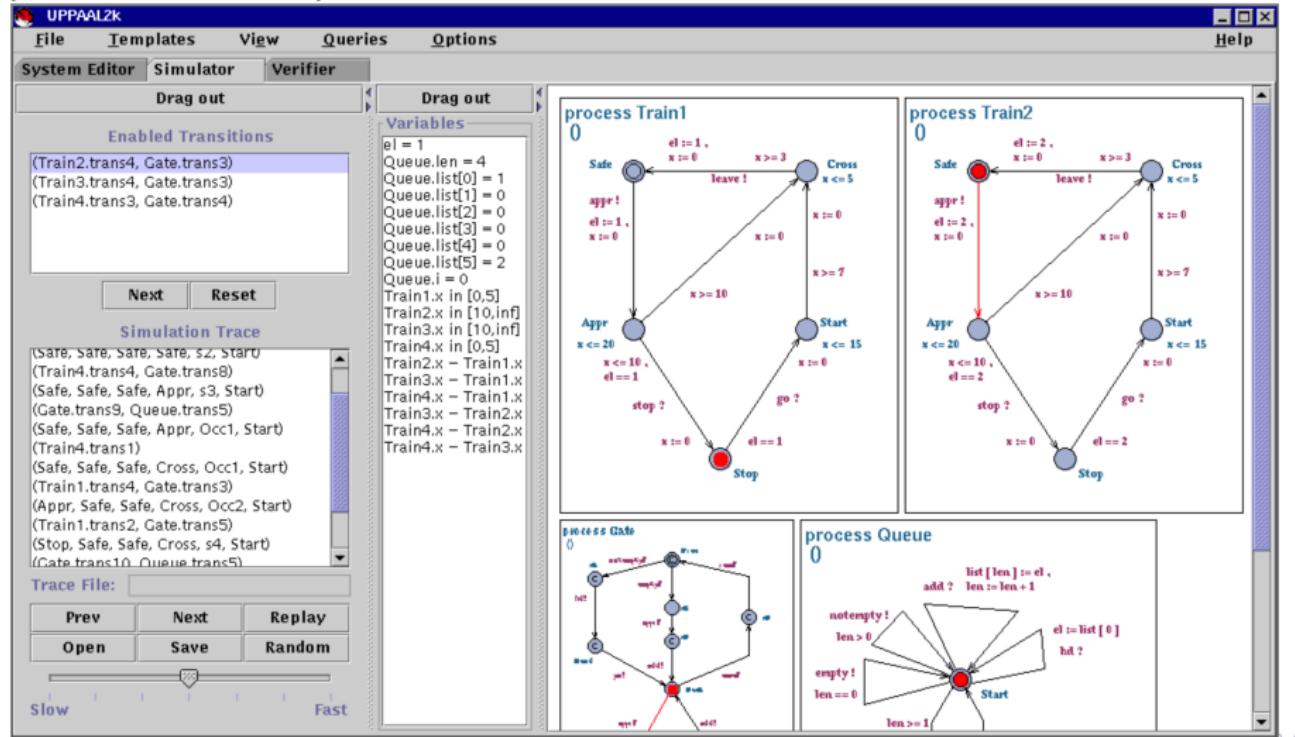
The Uppaal graphical editor (reminder)

Allows modelling a system process as an automata. The circles and arrows represent system states and transitions. Time progress – within in a state



The Uppaal simulator (reminder)

Provides visualisation of possible dynamic behaviours. Also shows traces (counter-examples) generated by the model checker



The Uppaal model checker (reminder)

Allows to formulate and verify invariant/safety and reachability/liveness system properties by exploring the system state space

The screenshot shows the UPPAAL2K interface with the following components:

- Menu Bar:** File, Templates, View, Queries, Options, Help.
- Toolbar:** System Editor, Simulator, Verifier.
- Overview Panel:** Displays a list of properties (P0 to P7) with their formulas and status indicators (green, red, grey).
 - P0: $E \leftrightarrow \text{Gate.Occ1}$ (Green)
 - P1: $E \leftrightarrow \text{Train1.Cross}$ (Green)
 - P2: $E \leftrightarrow \text{Train2.Cross}$ (Green)
 - P3: $E \leftrightarrow (\text{Train1.Cross} \text{ and } \text{Train2.Stop})$ (Red)
 - P4: $E \leftrightarrow (\text{Train1.Cross} \text{ and } \text{Train2.Stop} \text{ and } \text{Train3.Stop} \text{ and } \text{Train4.Stop})$ (Green)
 - P5: $E \leftrightarrow (\text{Train1.Cross} \text{ and } \text{Train2.Stop} \text{ and } \text{Train3.Stop} \text{ and } \text{Train4.Stop})$ (Green)
 - P6: $E \leftrightarrow \text{not}(\text{Train1.Cross} \text{ or } \text{Train2.Cross} \text{ or } \text{Train3.Cross} \text{ or } \text{Train4.Cross})$ (Grey)
 - P7: $A[] \text{not}(\text{Train1.Cross} \text{ and } (\text{Train2.Cross} \text{ or } \text{Train3.Cross} \text{ or } \text{Train4.Cross})) \backslash \text{or}(\text{Train2.Cross} \text{ and } (\text{Train3.Cross} \text{ or } \text{Train4.Cross})) \backslash \text{or}(\text{Train3.Cross} \text{ and } (\text{Train2.Cross} \text{ or } \text{Train4.Cross}))$ (Green)
- Model Check Buttons:** Insert, Remove, Comments.
- Query Panel:** Displays the query $E \leftrightarrow \text{Gate.Occ1}$.
- Comment Panel:** Displays the comment "Gate can receive (and store in queue) msg's from approaching trains."
- Status Panel:** Displays the status "Property is satisfied." followed by the formula $E \leftrightarrow (\text{Train1.Cross} \text{ and } \text{Train2.Stop} \text{ and } \text{Train3.Stop} \text{ and } \text{Train4.Stop})$.

Case study: satellite data processing unit

- A real example: the European Space Agency (ESA) mission BepiColombo, sending an orbiter (spacecraft) to Mercury to collect scientific data
- Data processing unit (DPU): the core part of the orbiter, operating scientific instruments and producing scientific data
- Software written by the company Space Systems Finland
- Scientific research (including software modelling and verification) within the EU project Rodin
- Both Event-B (the Rodin platform) and Uppaal were used for that

Bepi Colombo DPU: description

- DPU consists of the core software and software of scientific instruments
- The core software communicates with Bepi Colombo by sending telecommands (TCs), while the instrument software produces telemetrics (TMs) with scientific data
- Moreover, the spacecraft regularly produces housekeeping/diagnostics data (HKs) as special kind of TMs
- Both incoming TCs and outgoing TMs are stored in the respective (circular) buffers

Bepi Colombo DPU: description

- The incoming TCs are put into the TC buffer (if it is not full)
- Each TC should be validated (checked for correct data according to the standard) before being handled
- Once validated, a TC is forwarded to the instrument software for execution (e.g., changing the operational mode, managing the memory, requesting a particular type of scientific data, etc.)
- A generated TM is put into the TM buffer (if it is not full) for sending back
- A separate housekeeping unit is responsible for regularly generating diagnostic reports and putting them into the TM buffer

The system parameters affecting the overall system performance:

- the size of circular buffers for storing TCs and TMs;
- the worst execution time for TC validation;
- the worst execution time for the instrument responding to the forwarded TC;
- the period of the process regularly generating housekeeping data returned as additional TM;
- the maximal delay before the generated TM is delivered
- ...

Modelling the Bepi Colombo system in Uppaal

The Bepi Colombo system in the Uppaal notation:

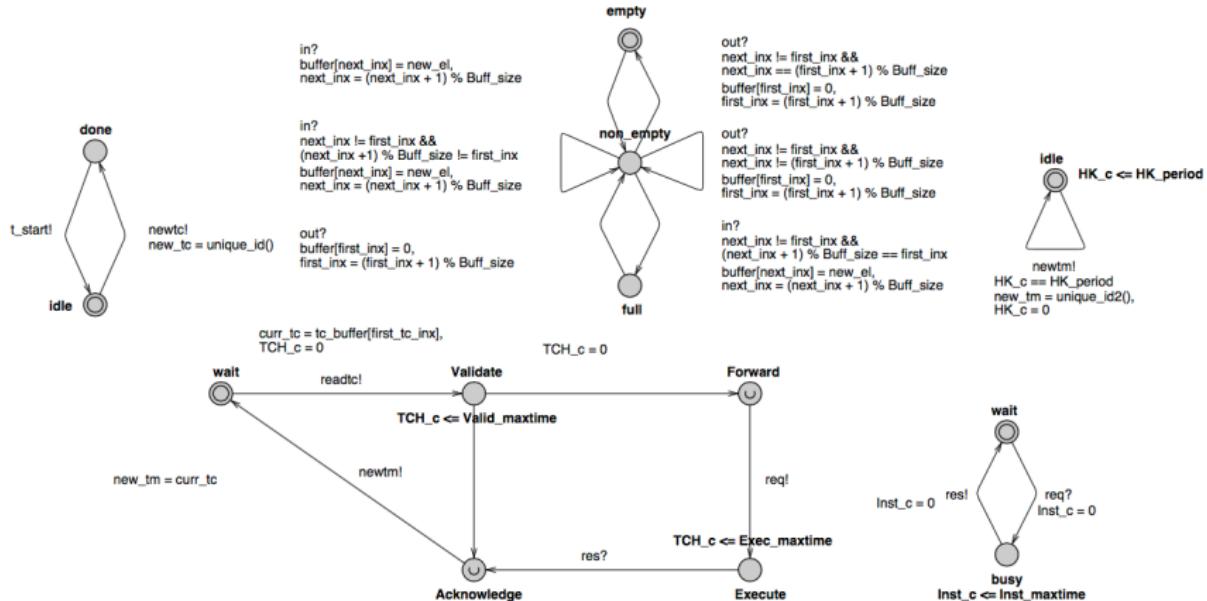


Fig. 2. The BepiColombo process view model Uppaal notation.

Modelling elements: semantics

Annotations and their explanations:

- State transitions (arrows): $x?$ and $x!$ actions that are synchronised with other processes
- State transitions (arrows): conditions/guards enabling a transition
- State transitions (arrows): $v = \text{expr}$ assignments to model variables
- States (circles): conditions or probabilistic rates (frequencies) on clock variables expressing time constraints for staying in that state
- ...

Bepi Colombo DPU: properties to check

- For this system, we are interesting in two kinds of properties to verify (model check)
- First, we need to ensure that, for any received TC, the corresponding TM (indicating either success or failure) is eventually returned
- Second, we need to check that the worst execution time of TC handling does not exceed the pre-defined limit (depending on various system parameters such as the buffer size or the period of housekeeping data generation)

Some examples of checked timing properties (reminder)

- $E<> p$: there exists an execution path where p eventually holds;
- $A[] p$: for all paths p always holds (invariant);
- $E[] p$: there is an execution path where p always holds;
- $A<> p$: for all paths p will eventually hold;
- $p \rightarrowtail q$: whenever p holds, q will eventually hold.

Bepi Colombo DPU: quantitative verification examples

- The first kind of properties can be expressed in Uppaal as, e.g.,:

```
(new_tc==1) --> (last_tm==1)
```

for the message id 1 (the property can be checked independently of concrete id values)

- The first kind of properties can be expressed in Uppaal as follows (using the values of clock variables):

```
A[] (last_tm == 1 && Observer1.stop) imply  
(Observer1.Obs c < upper_bound)
```

where $A[]$ means "Always, for any execution path", while Observer1 is a special process that starts the clock Observer1.Obs c , whenever a TC command with the id 1 is received, and stops it, once the corresponding TM is returned

Quantitative assessment

Investigating how the HK production affects the overall TC handling time

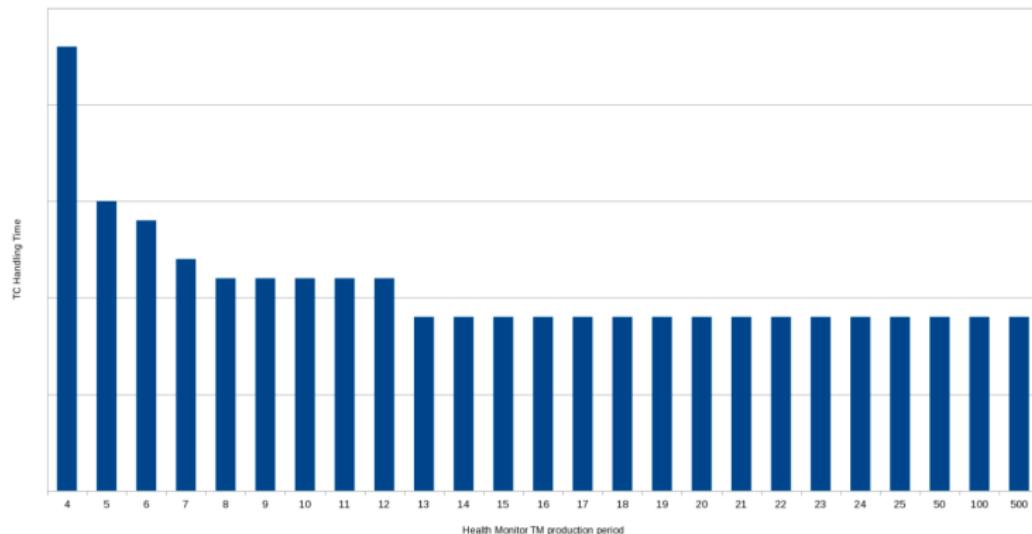


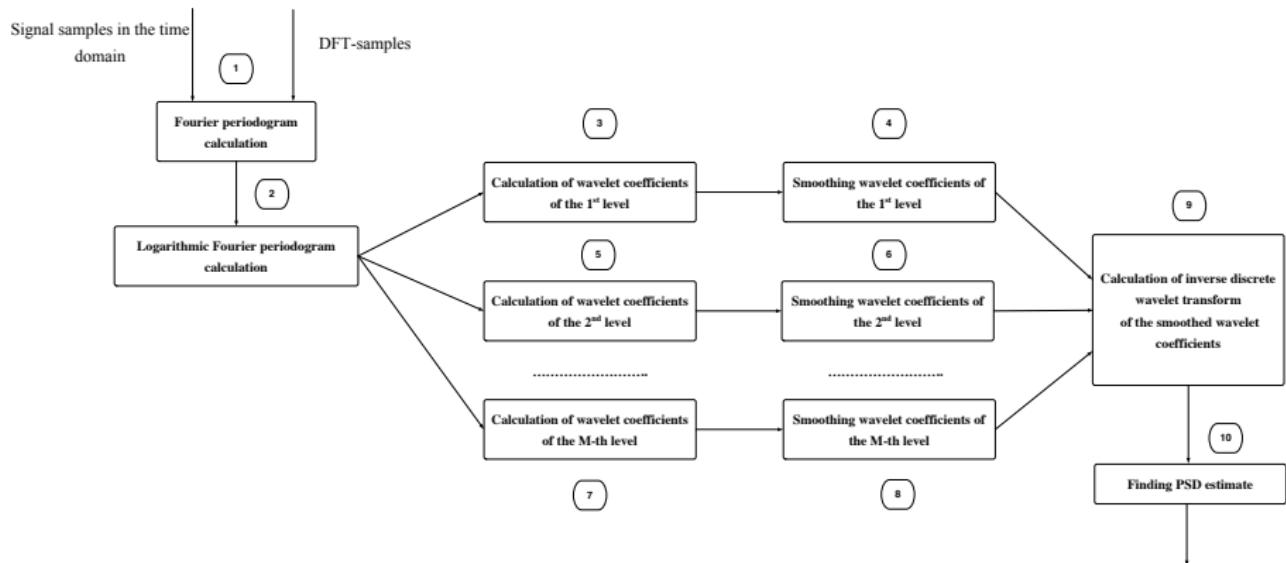
Fig. 3. The relationship between the period of TM production by *Health Monitor* (X-axis) and worst TC handling time (Y-axis). For periods less than 4, the system is potentially in livelock. X-axis numbers are the multiples of a base measuring constant – the TC decoding time.

Another case study: signal processing in a distributed system

- Signal processing of acoustic signals in a dynamic distributed system with a number of data processing workers (servers), based on the pre-defined calculation algorithm
- Signal preprocessing from multi-channel observations (recorded sea sounds from reception hydrophones), filtering noises and other interferences
- Purpose of the project: to formally develop and quantitatively evaluate software architectures that most suitable for effective application of the proposed signal processing algorithm
- Again, both Event-B (the Rodin platform) and Uppaal (statistical version this time) were used for that

Signal processing

Algorithm structure:



Signal processing: system architecture

- One master component coordinating the overall execution of the algorithm
- Many worker components that are capable of executing specific calculations assigned by the master component
- Some algorithm phases involve (if possible) parallel computations by several workers. The optimal number of parallel computations that a specific phase can be split into is known beforehand
- The availability of workers is dynamic (i.e., some components can be busy/unavailable, they also can dynamically fail or recover)

Signal processing: system parameters

System parameters as Uppaal textual declarations (in a separate file):

```
// Place global declarations here.

const int N = 10;           // number of components
typedef int[0,N-1] id_comp;

const int M = 16;           // number of possible parallel computations
typedef int [0,M-1] id_tasks;

const int TASK_TIME = 34;    // task calculation time
const int INIT_DELAY = 2;    // initial delay for receiving data
const int PERIOD = 5;       // monitoring period
const int MIN_DELAY = 1;     // minimal communication delay

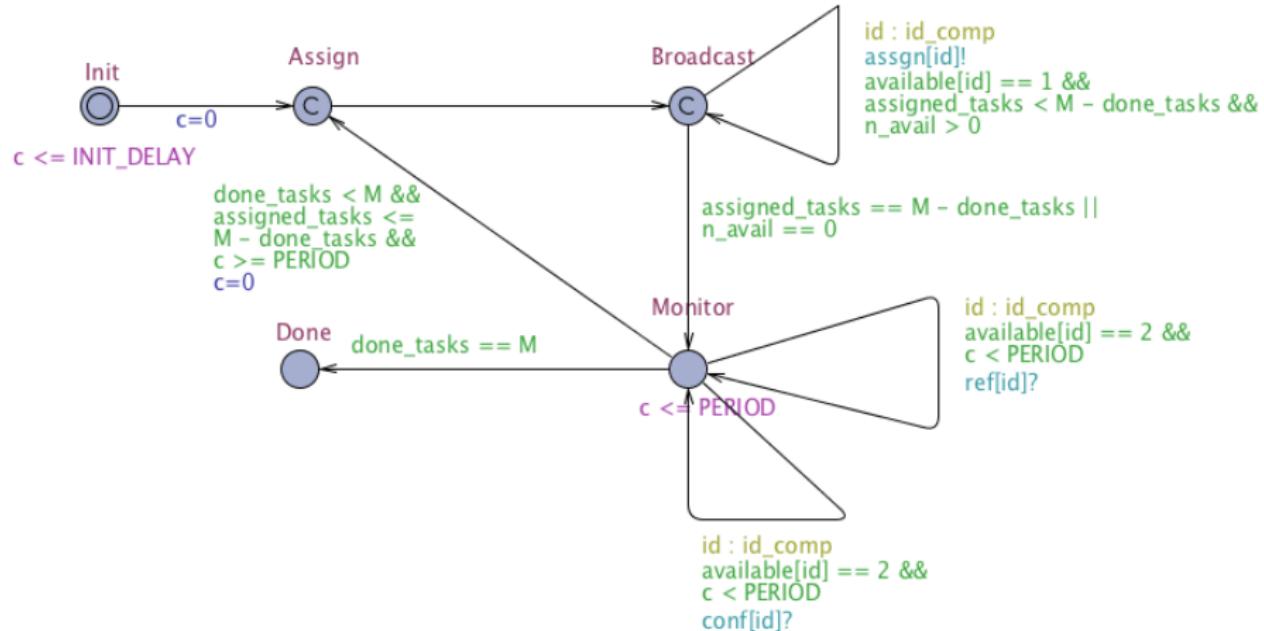
const int pw_aa = 95;       // probab. weight of component staying available
const int pw_au = 5;         // probab. weight of component becoming unavailable
const int pw_ua = 15;        // probab. weight of component becoming available
const int pw_uu = 85;        // probab. weight of component staying unavailable
const int pw_suc = 95;       // probab. weight of component finishing a given task
const int pw_ref = 5;        // probab. weight of component failing a given task

broadcast chan assgn[N];   // channel for broadcasting task assignments
broadcast chan conf[N];    // channel for confirming task completion
broadcast chan ref[N];    // channel for refusing task execution

int[0,M] done_tasks = 0;    // number of completed tasks
int[0,M] assigned_tasks = 0; // number of assigned tasks
int[0,N] n_avail = 0;       // number of components available for task execution

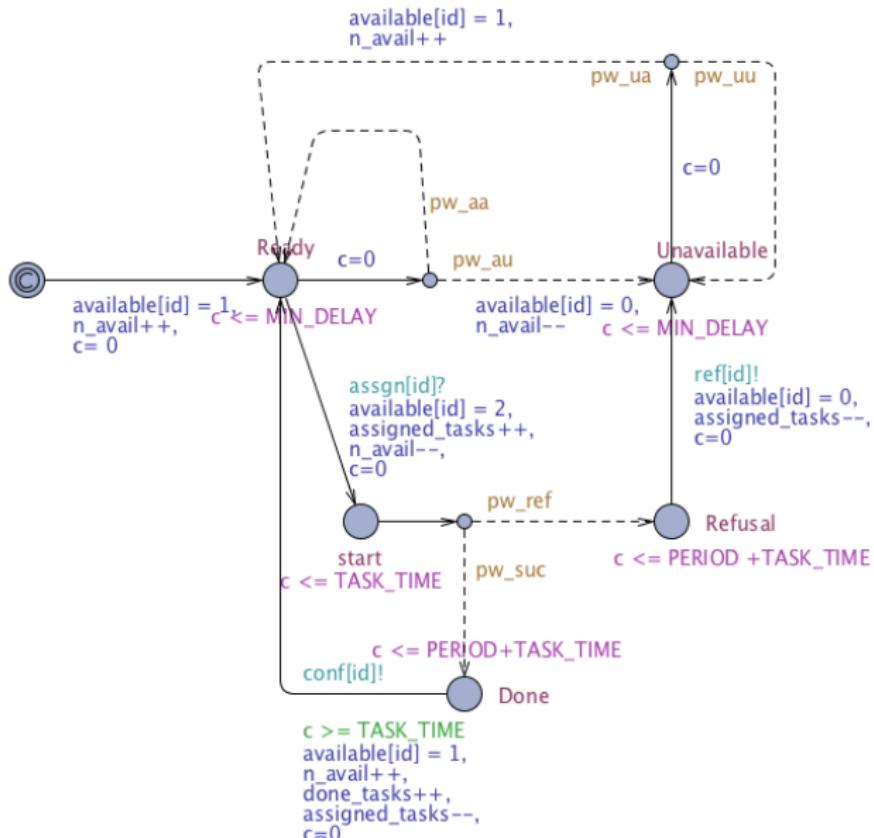
int[0,2] available[N] = {0,0,0,0,0,0,0,0,0,0}; // component availability status
```

Signal processing: the Master component



Here $\text{id} : \text{id_comp}$ is a component parameter (like ANY clause in Event-B)

Signal processing: the Worker component

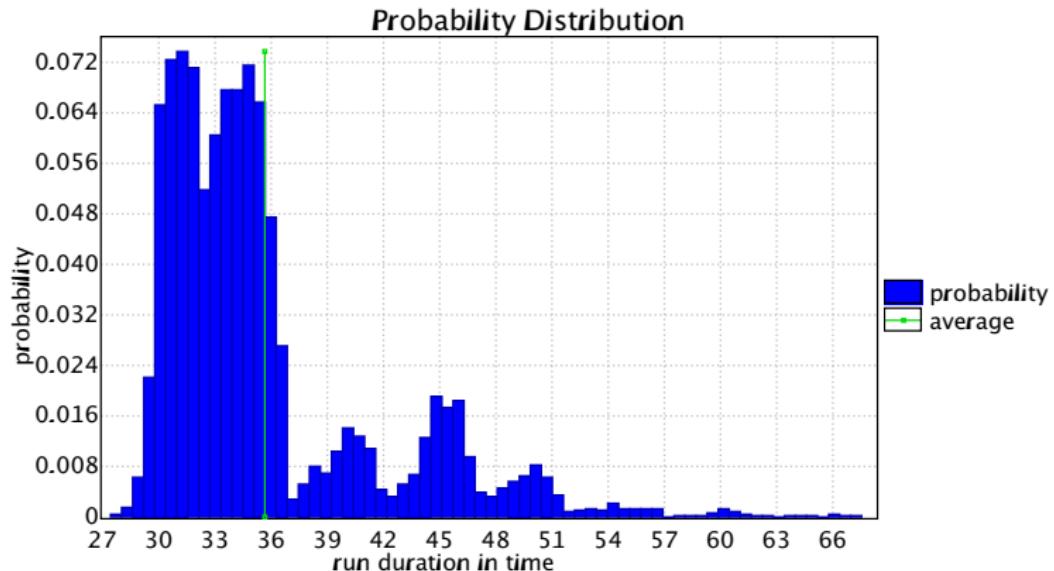


Signalling: quantitative verification examples

- A number of required time reachability properties, considering different value combinations for system parameters. All the verified properties are of the form:
 $\text{Pr } [\leq \text{time_bound}] (\text{<>} \text{ Master.Done})$
- The result is the probability that the Master component eventually reaches the state `Master.Done` (i.e., the state where all the parallel task calculations are successfully completed) within the given time bound
- Moreover, the obtained results can be graphically plotted to show probability distribution or cumulative probability distribution for different values of the clock

Signal processing

Probability distribution (for the data size 20 and 10 worker components):



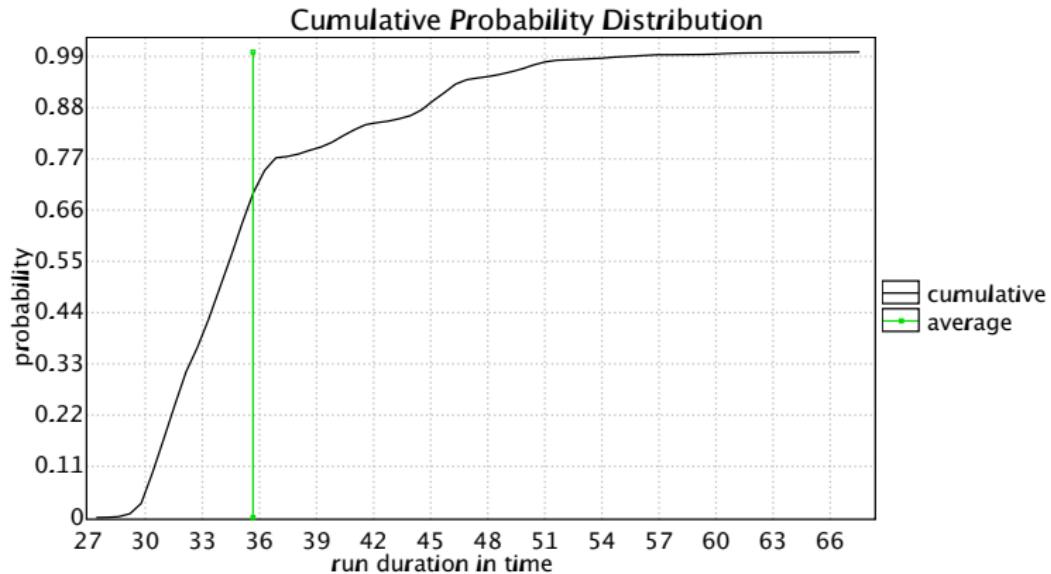
Runs: 4612 in total, 4609 displayed, 3 remaining.

Probability sums: 0.99935 displayed, 0.000650477 remaining.

Minimum, maximum, average: 27.4211, 67.5414, 35.6614.

Signal processing

Cumulative probability distribution (for the data size 20 and 10 worker components)::



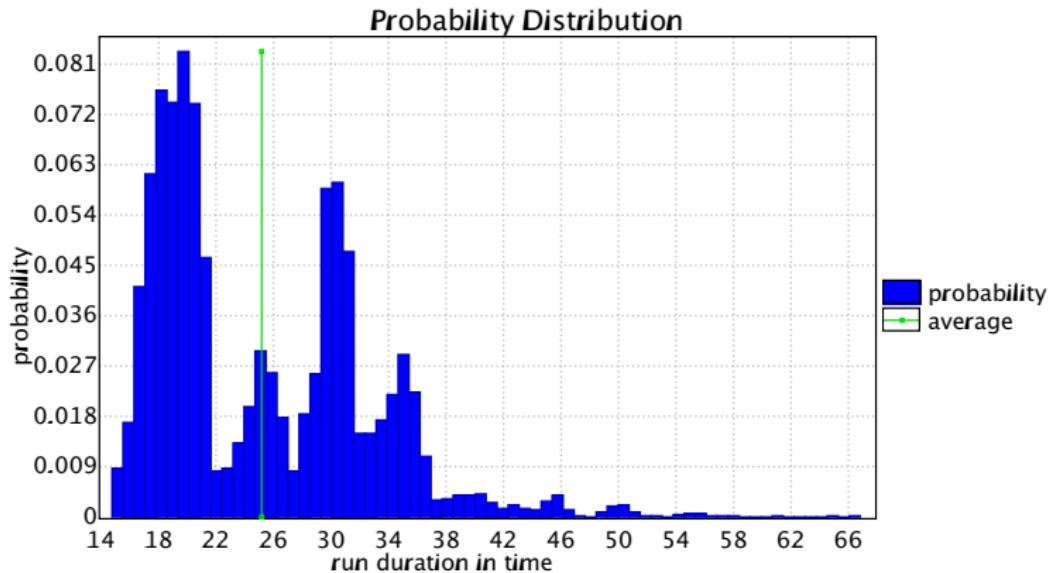
Runs: 4612 in total, 4609 displayed, 3 remaining.

Probability sums: 0.99935 displayed, 0.000650477 remaining.

Minimum, maximum, average: 27.4211, 67.5414, 35.6614.

Signal processing

Probability distribution (for the data size 20 and 16 worker components)::



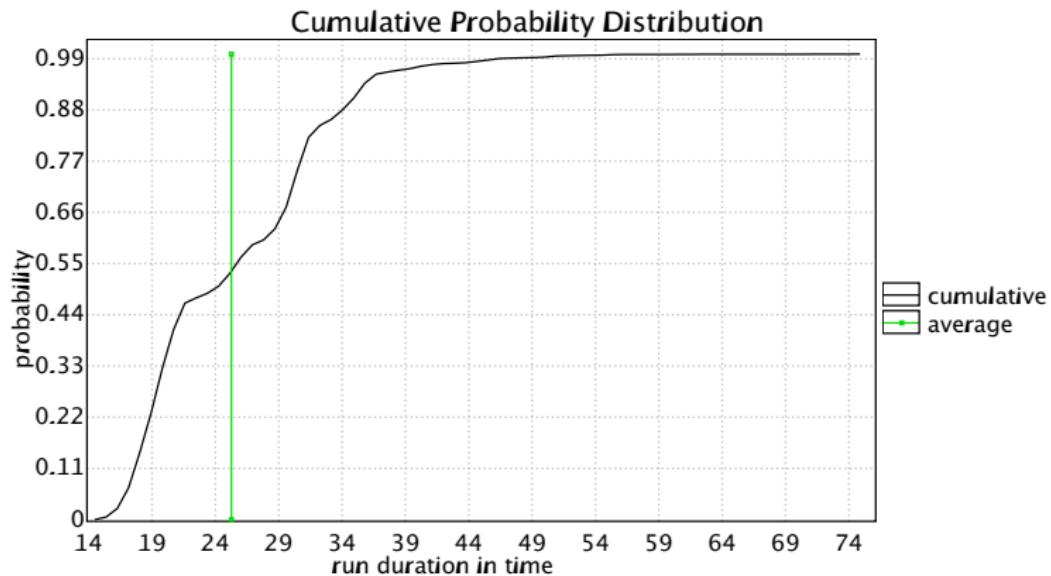
Runs: 4612 in total, 4612 displayed, 0 remaining.

Probability sums: 1 displayed, 0 remaining.

Minimum, maximum, average: 14.7551, 66.8134, 25.1675.

Signal processing

Cumulative probability distribution (for the data size 20 and 16 worker components)::



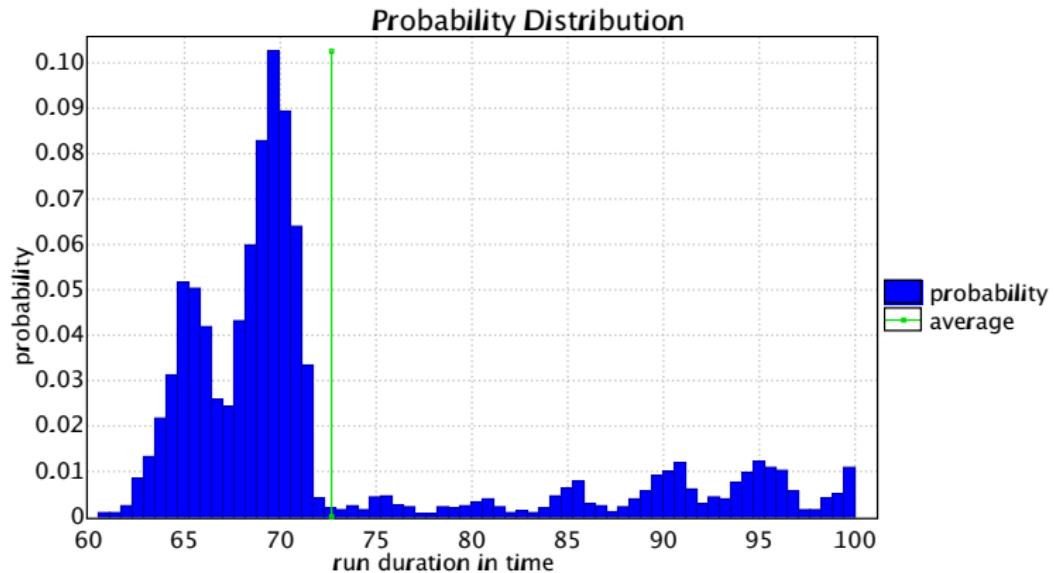
Runs: 4612 in total, 4612 displayed, 0 remaining.

Probability sums: 1 displayed, 0 remaining.

Minimum, maximum, average: 14.484, 74.8756, 25.2627.

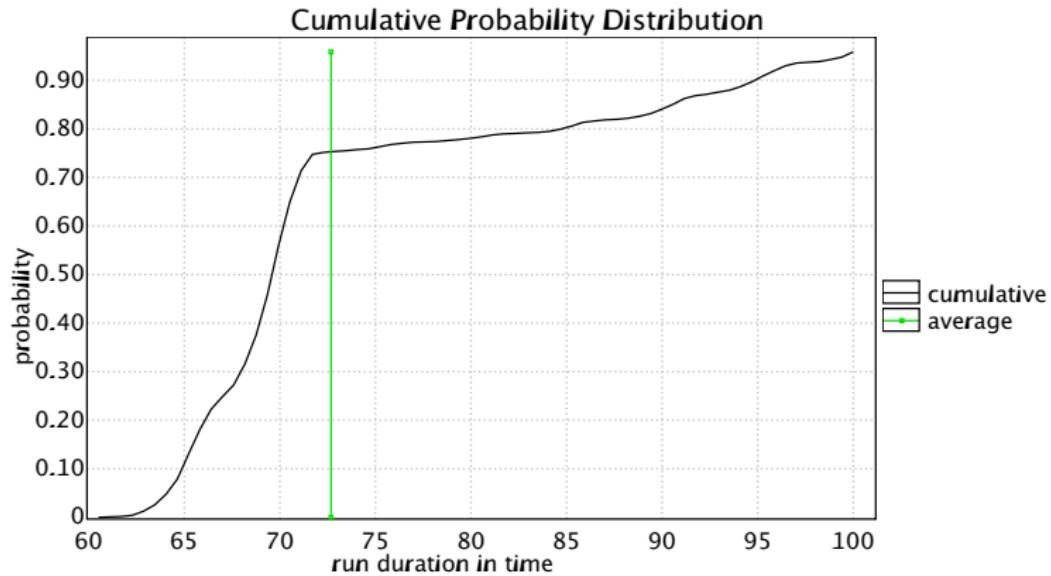
Signal processing

Probability distribution (for the data size 40 and 10 worker components)::



Signal processing

Cumulative probability distribution (for the data size 40 and 10 worker components)::



Runs: 4612 in total, 4422 displayed, 190 remaining.

Probability sums: 0.958803 displayed, 0.0411969 remaining.

Minimum, maximum, average: 60.5077, 99.9952, 72.6749.

Signal processing: analysis of the optimal system architecture

- Further human expertise needed to choose the best architecture from these generated basic analysis data
- Possible "rules of thumb" (heuristics): probability distribution should be similar to the normal probabilistic one, with average time being the only main peak. Or, cumulative distribution should reach 0.99 and plateau from there as soon as possible.
- Different artificial intelligence and machine learning techniques are also applicable here

Lecture 15: Outline

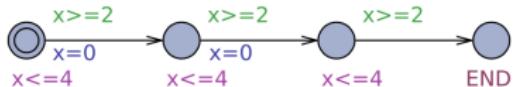
- Statistical model checking (reminder)
- A case study with Uppaal-SMC
- Course wrap-up. Exam information

The Uppaal SMC tool (reminder)

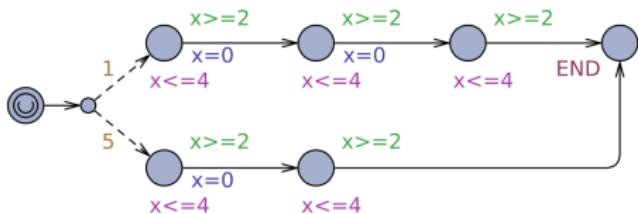
- Recently, the Uppaal tool was extended to support statistical model checking (SMC), where properties are checked with some statistical confidence (probabilities)
- The Uppaal language for modelling real-time systems has also been extended with probabilistic transition branching and clock rates (probabilistic distribution)
- The exhaustive model checking of the system state space is replaced with simulating the system behaviour for some number of "system runs" and accumulating the statistical results
- Timed automata are thus augmented to become stochastic timed automata

The Uppaal SMC tool (cont.)

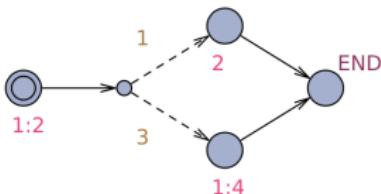
The language is extended with probabilistic transition branching (in 2nd and 3rd diagrams) and clock rates (in 3rd diagram)



(a) $A_1.$

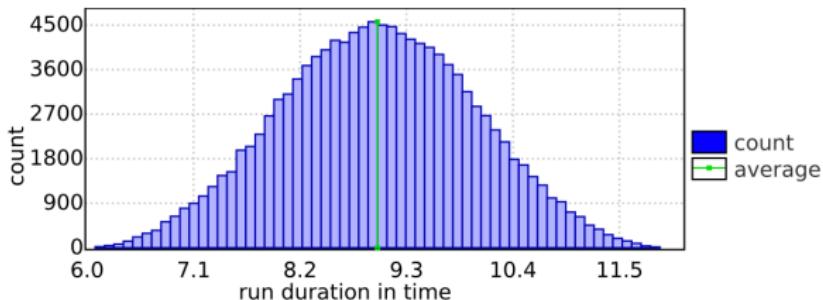


(b) $A_2.$

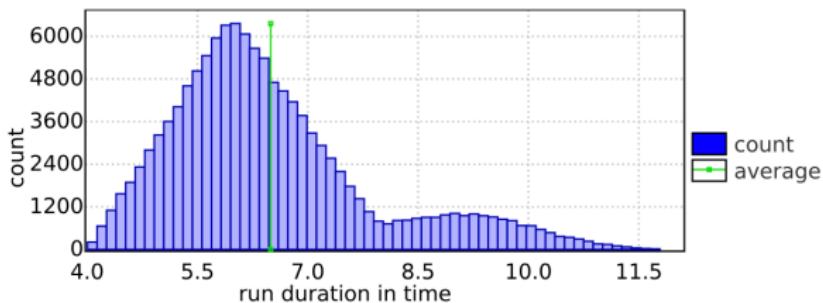


The Uppaal SMC tool (cont.)

Visualisation of distribution of reachability time



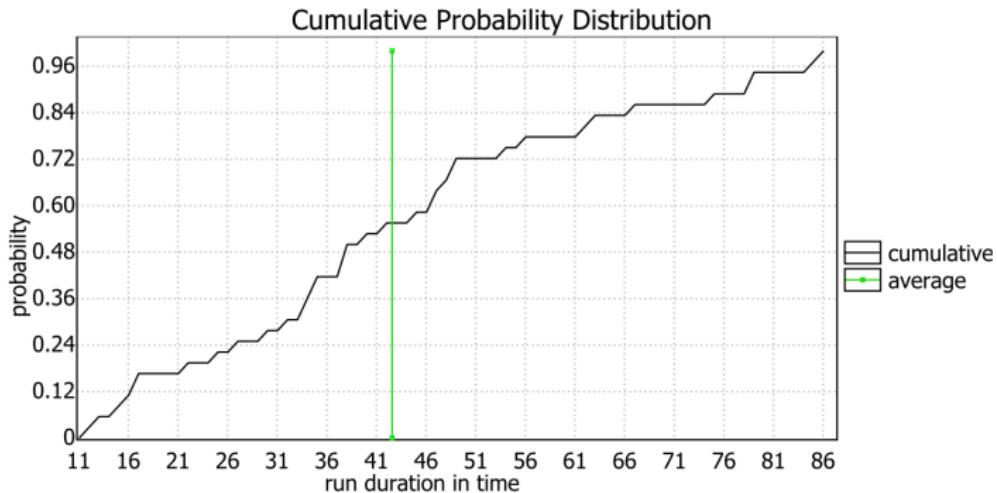
(a) A_1 arrival to **END**.



(b) A_2 arrival to **END**.

The Uppaal SMC tool (cont.)

The cumulative probability distribution of reaching a desired state within the given time



Parameters: $\alpha=0.05$, $\varepsilon=0.05$, bucket width=1, bucket count=75

Runs: 36 in total, 36 (100%) displayed, 0 (0%) remaining

Span of displayed sample: [11.0447, 85.2899]

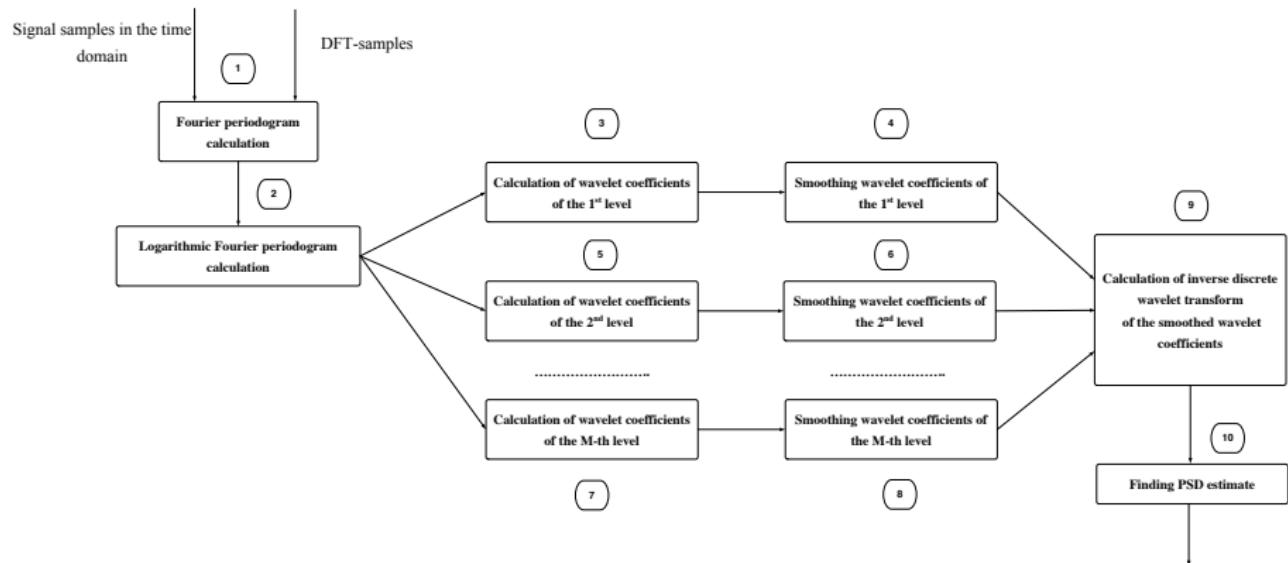
Mean of displayed sample: 42.5735 ± 7.10607 (95% CI)

Another case study: signal processing in a distributed system

- Signal processing of acoustic signals in a dynamic distributed system with a number of data processing workers (servers), based on the pre-defined calculation algorithm
- Signal preprocessing from multi-channel observations (recorded sea sounds from reception hydrophones), filtering noises and other interferences
- Purpose of the project: to formally develop and quantitatively evaluate software architectures that most suitable for effective application of the proposed signal processing algorithm
- Again, both Event-B (the Rodin platform) and Uppaal (statistical version this time) were used for that

Signal processing

Algorithm structure:



Signal processing: system architecture

- One master component coordinating the overall execution of the algorithm
- Many worker components that are capable of executing specific calculations assigned by the master component
- Some algorithm phases involve (if possible) parallel computations by several workers. The optimal number of parallel computations that a specific phase can be split into is known beforehand
- The availability of workers is dynamic (i.e., some components can be busy/unavailable, they also can dynamically fail or recover)

Signal processing: system parameters

System parameters as Uppaal textual declarations (in a separate file):

```
// Place global declarations here.

const int N = 10;           // number of components
typedef int[0,N-1] id_comp;

const int M = 16;           // number of possible parallel computations
typedef int [0,M-1] id_tasks;

const int TASK_TIME = 34;    // task calculation time
const int INIT_DELAY = 2;    // initial delay for receiving data
const int PERIOD = 5;       // monitoring period
const int MIN_DELAY = 1;     // minimal communication delay

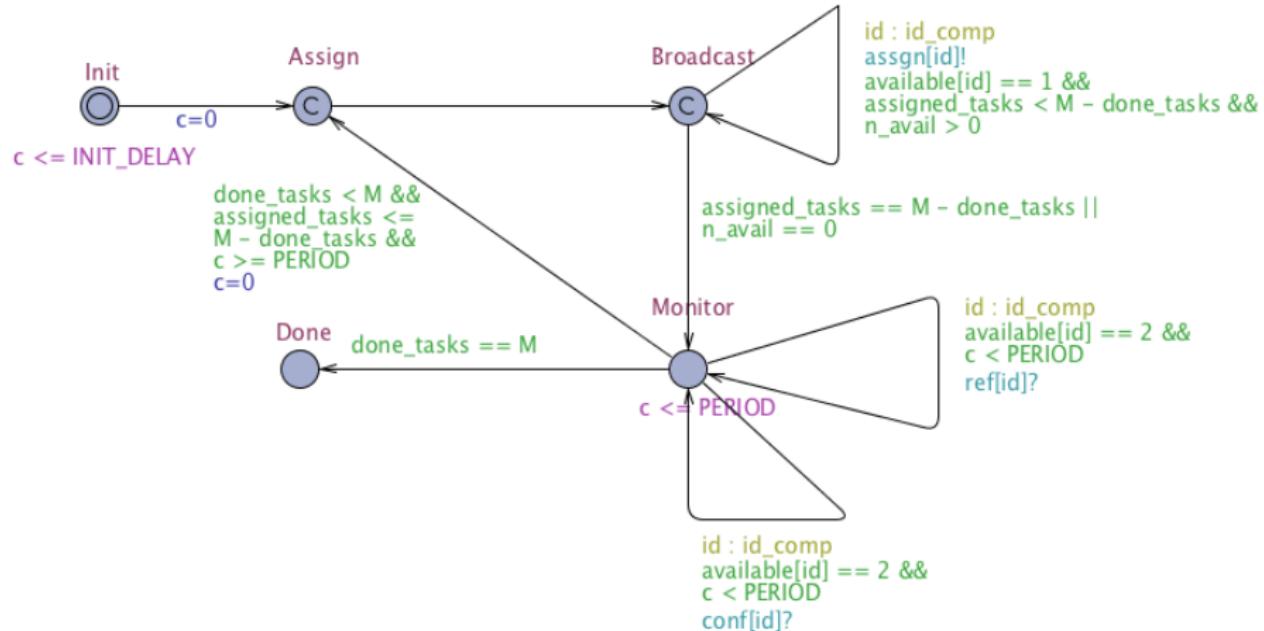
const int pw_aa = 95;       // probab. weight of component staying available
const int pw_au = 5;         // probab. weight of component becoming unavailable
const int pw_ua = 15;        // probab. weight of component becoming available
const int pw_uu = 85;        // probab. weight of component staying unavailable
const int pw_suc = 95;       // probab. weight of component finishing a given task
const int pw_ref = 5;        // probab. weight of component failing a given task

broadcast chan assgn[N];   // channel for broadcasting task assignments
broadcast chan conf[N];   // channel for confirming task completion
broadcast chan ref[N];   // channel for refusing task execution

int[0,M] done_tasks = 0;    // number of completed tasks
int[0,M] assigned_tasks = 0; // number of assigned tasks
int[0,N] n_avail = 0;      // number of components available for task execution

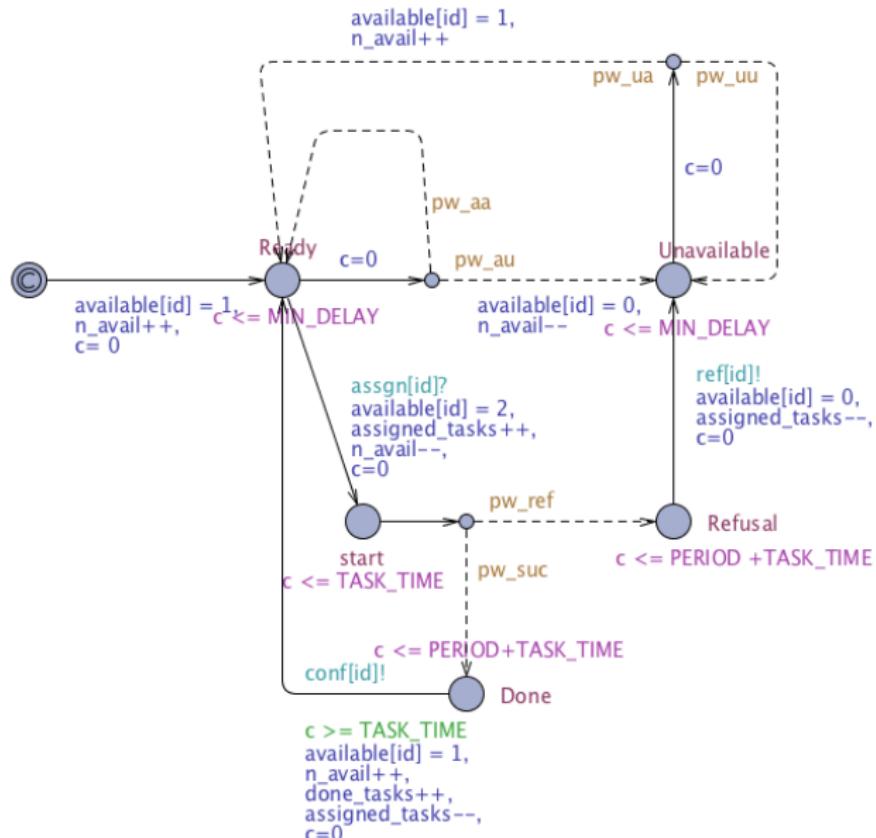
int[0,2] available[N] = {0,0,0,0,0,0,0,0,0,0}; // component availability status
```

Signal processing: the Master component



Here $\text{id} : \text{id_comp}$ is a component parameter (like ANY clause in Event-B)

Signal processing: the Worker component

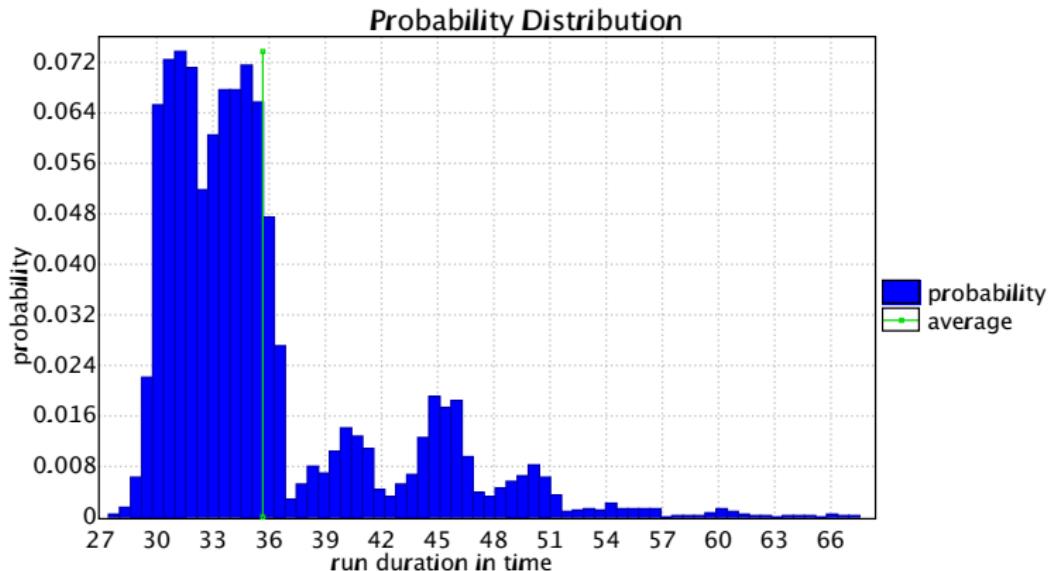


Signalling: quantitative verification examples

- A number of required time reachability properties, considering different value combinations for system parameters. All the verified properties are of the form:
 $\text{Pr } [\leq \text{time_bound}] (\text{Master.Done})$
- The result is the probability that the Master component eventually reaches the state `Master.Done` (i.e., the state where all the parallel task calculations are successfully completed) within the given time bound
- Moreover, the obtained results can be graphically plotted to show probability distribution or cumulative probability distribution for different values of the clock

Signal processing

Probability distribution (for the data size 20 and 10 worker components):



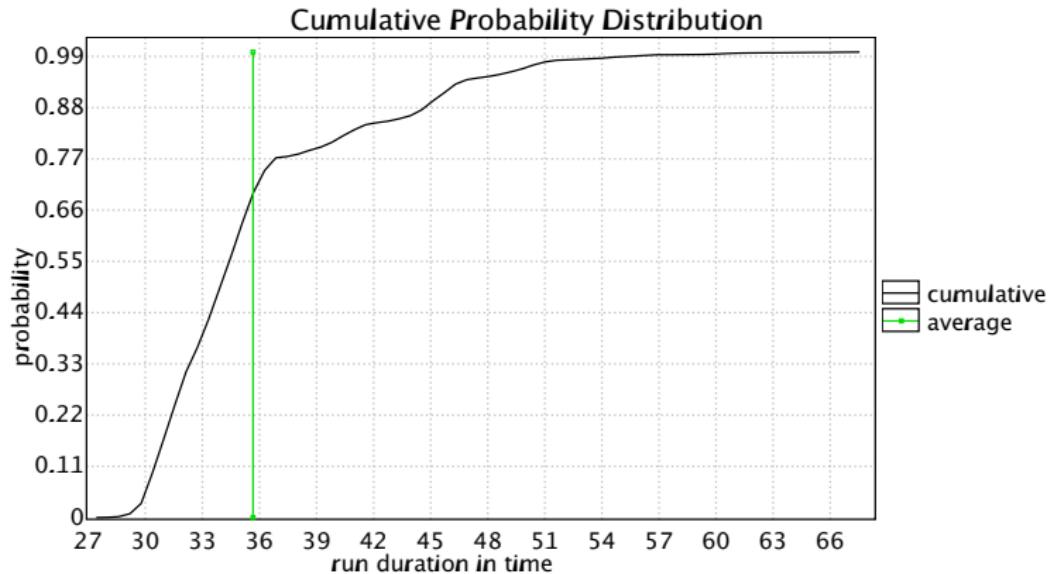
Runs: 4612 in total, 4609 displayed, 3 remaining.

Probability sums: 0.99935 displayed, 0.000650477 remaining.

Minimum, maximum, average: 27.4211, 67.5414, 35.6614.

Signal processing

Cumulative probability distribution (for the data size 20 and 10 worker components)::



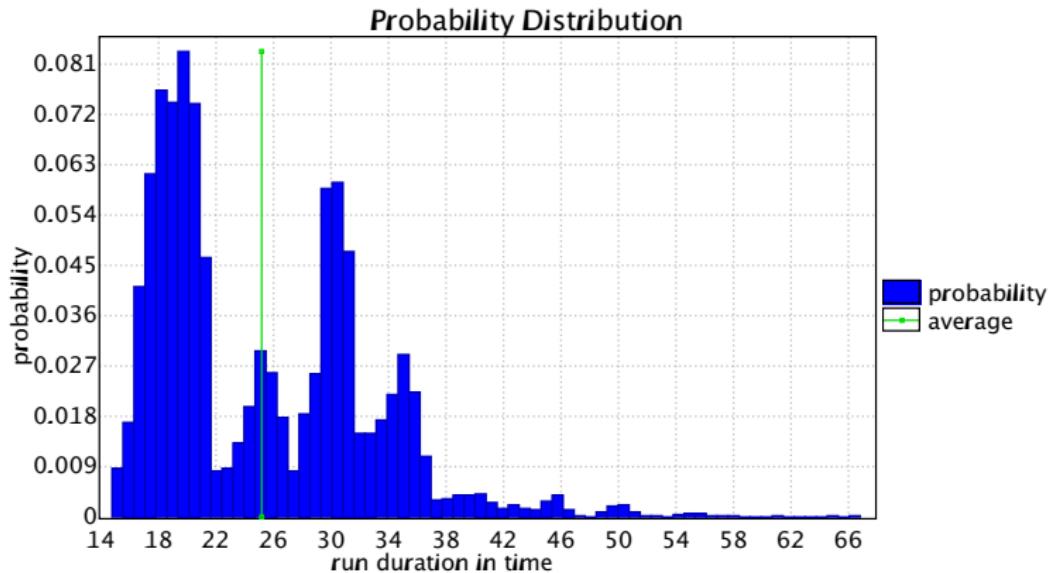
Runs: 4612 in total, 4609 displayed, 3 remaining.

Probability sums: 0.99935 displayed, 0.000650477 remaining.

Minimum, maximum, average: 27.4211, 67.5414, 35.6614.

Signal processing

Probability distribution (for the data size 20 and 16 worker components)::



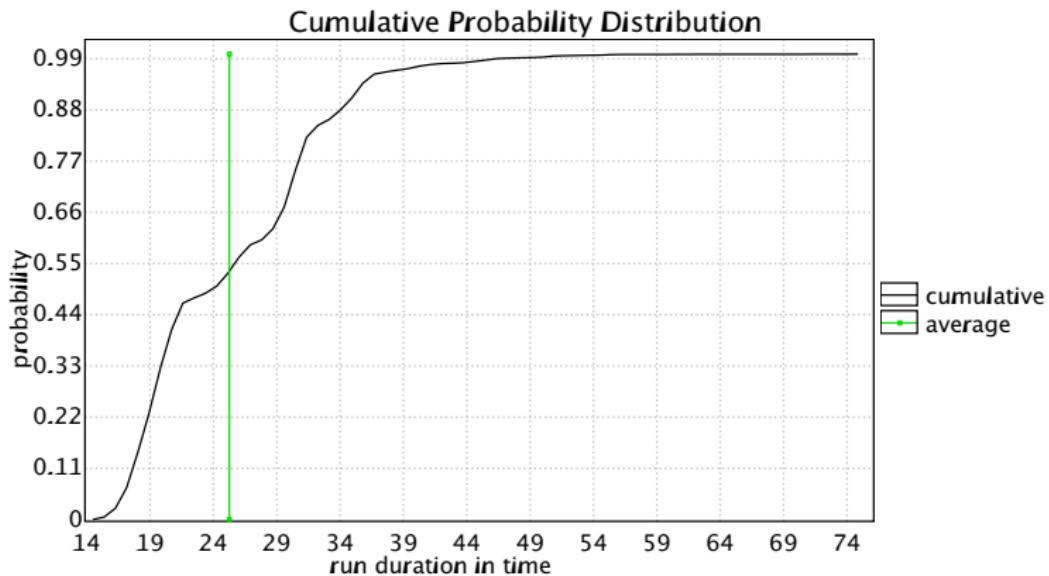
Runs: 4612 in total, 4612 displayed, 0 remaining.

Probability sums: 1 displayed, 0 remaining.

Minimum, maximum, average: 14.7551, 66.8134, 25.1675.

Signal processing

Cumulative probability distribution (for the data size 20 and 16 worker components)::



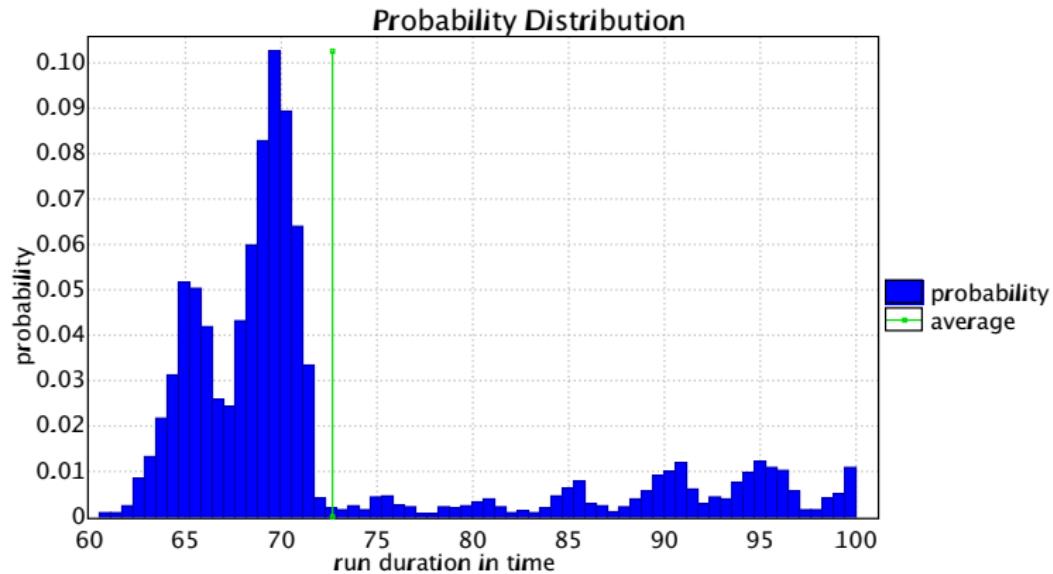
Runs: 4612 in total, 4612 displayed, 0 remaining.

Probability sums: 1 displayed, 0 remaining.

Minimum, maximum, average: 14.484, 74.8756, 25.2627.

Signal processing

Probability distribution (for the data size 40 and 10 worker components)::



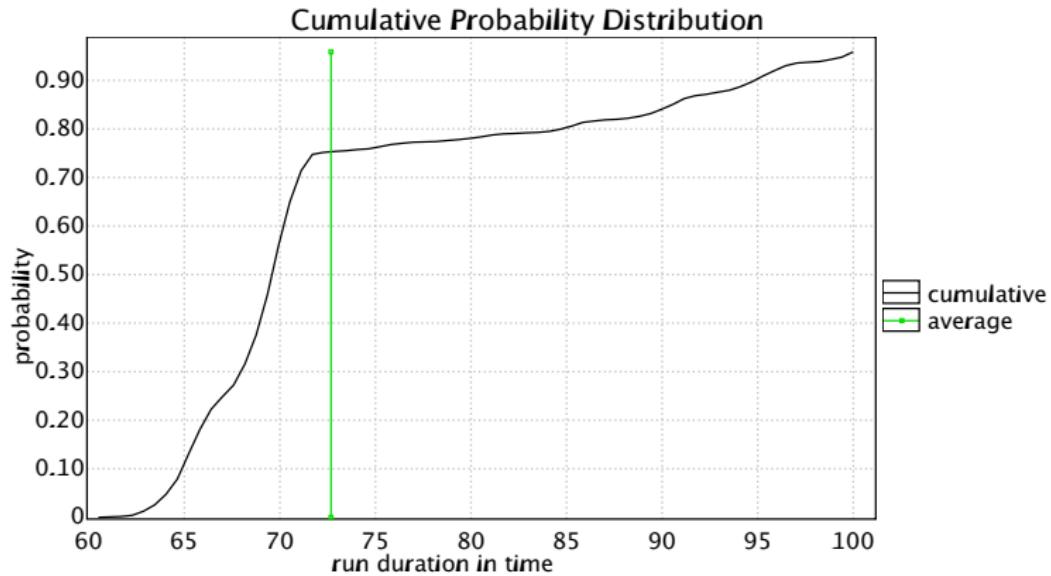
Runs: 4612 in total, 4422 displayed, 190 remaining.

Probability sums: 0.958803 displayed, 0.0411969 remaining.

Minimum, maximum, average: 60.5077, 99.9952, 72.6749.

Signal processing

Cumulative probability distribution (for the data size 40 and 10 worker components)::



Runs: 4612 in total, 4422 displayed, 190 remaining.

Probability sums: 0.958803 displayed, 0.0411969 remaining.

Minimum, maximum, average: 60.5077, 99.9952, 72.6749.

Signal processing: analysis of the optimal system architecture

- Further human expertise needed to choose the best architecture from these generated basic analysis data
- Possible "rules of thumb" (heuristics): probability distribution should be similar to the normal probabilistic one, with average time being the only main peak. Or, cumulative distribution should reach 0.99 and plateau from there as soon as possible.
- Different artificial intelligence and machine learning techniques are also applicable here

Wrapping-up of the course

- Extra time for exercise presentations: next Wednesday, June 5th, 18.00 – 21.00, 421 Didlaukio (Scheduler slot reservation in Moodle)
- Exam: 07.06.2018, 16.00 – 18.00, 103 Didlaukio
- No more than two tasks: defining data in a context or writing an invariant or an operation (event) for a given partial model, creating an Event-B model from system requirements
- Using printed lecture slides or any written notes is allowed