Programų sistemų 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: Thuesdays 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 an 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 nonderterminism due to inability to control),
  - model parallel execution (using the interleaving semantics)

Formal models: examples

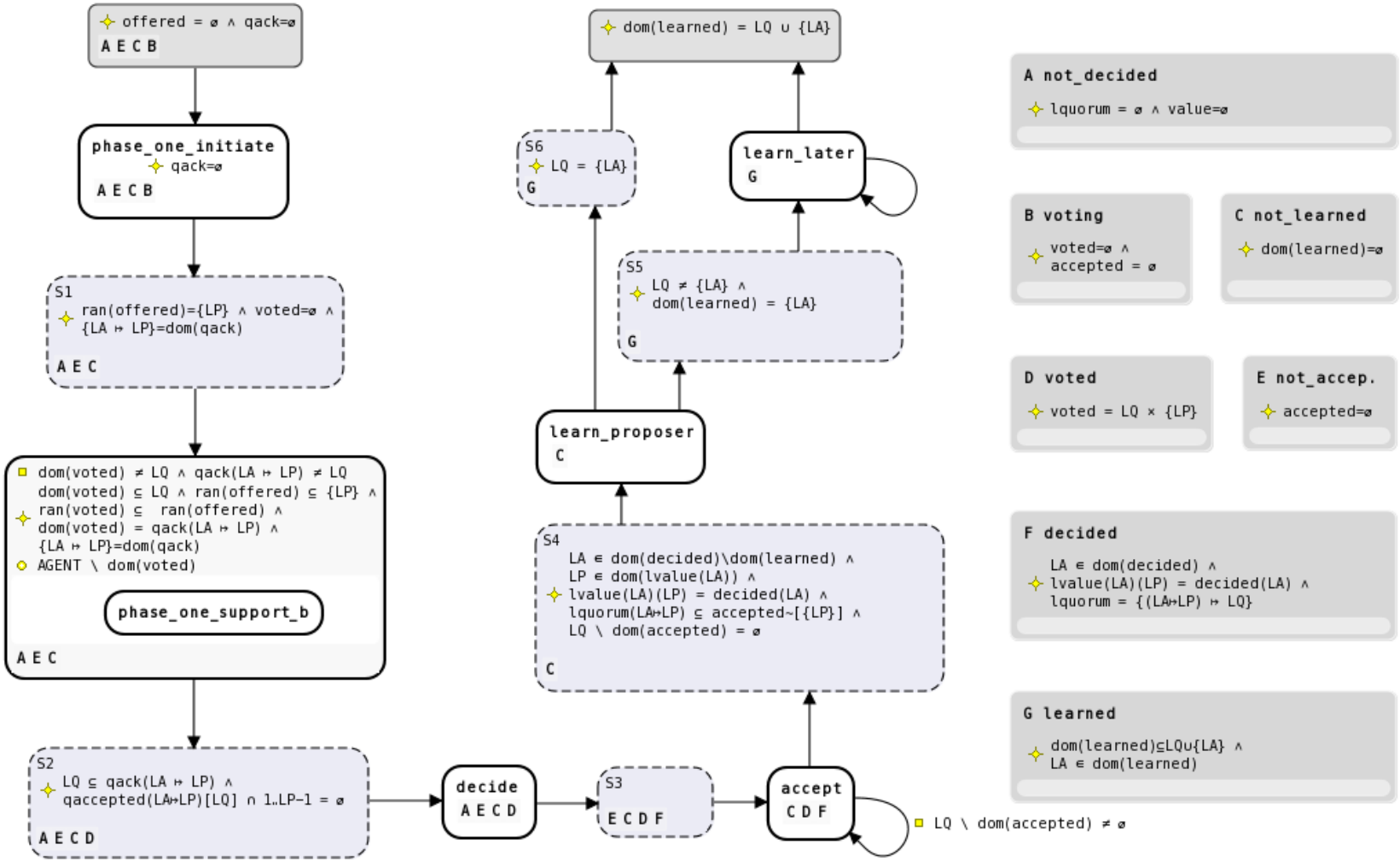
- An Event-B model: textual description of state-based component, its state (variables) and transitions (reactions)

Context *RailwayCrossing_ctx*
$POS\_SET = \{0, CRP, SRP, SRS, DS\}$, // $0 < CRP < SRP < SRS < DS$
$PHASES = \{Env, Train, Crossing\}$
$BAR\_POS = \{Open, Closed\}$

Machine *RailwayCrossing*
Variables $train\_pos, phase, emrg\_brakes, bar_1, bar_2$
Invariants $\cdots$
Events $\cdots$
$UpdatePosition_1 \mathrel{\widehat{=}}$
when
$phase = Env \wedge train\_pos < DS \wedge emrg\_brakes = FALSE$
then
$train\_pos := \min(\{p \mid p \in POS\_SET \wedge p > train\_pos\}) \,||\, phase := Train$
end
$UpdatePosition_2 \mathrel{\widehat{=}}$
when
$phase = Env \wedge (train\_pos = DS \vee emrg\_brakes = TRUE)$
then
skip
end

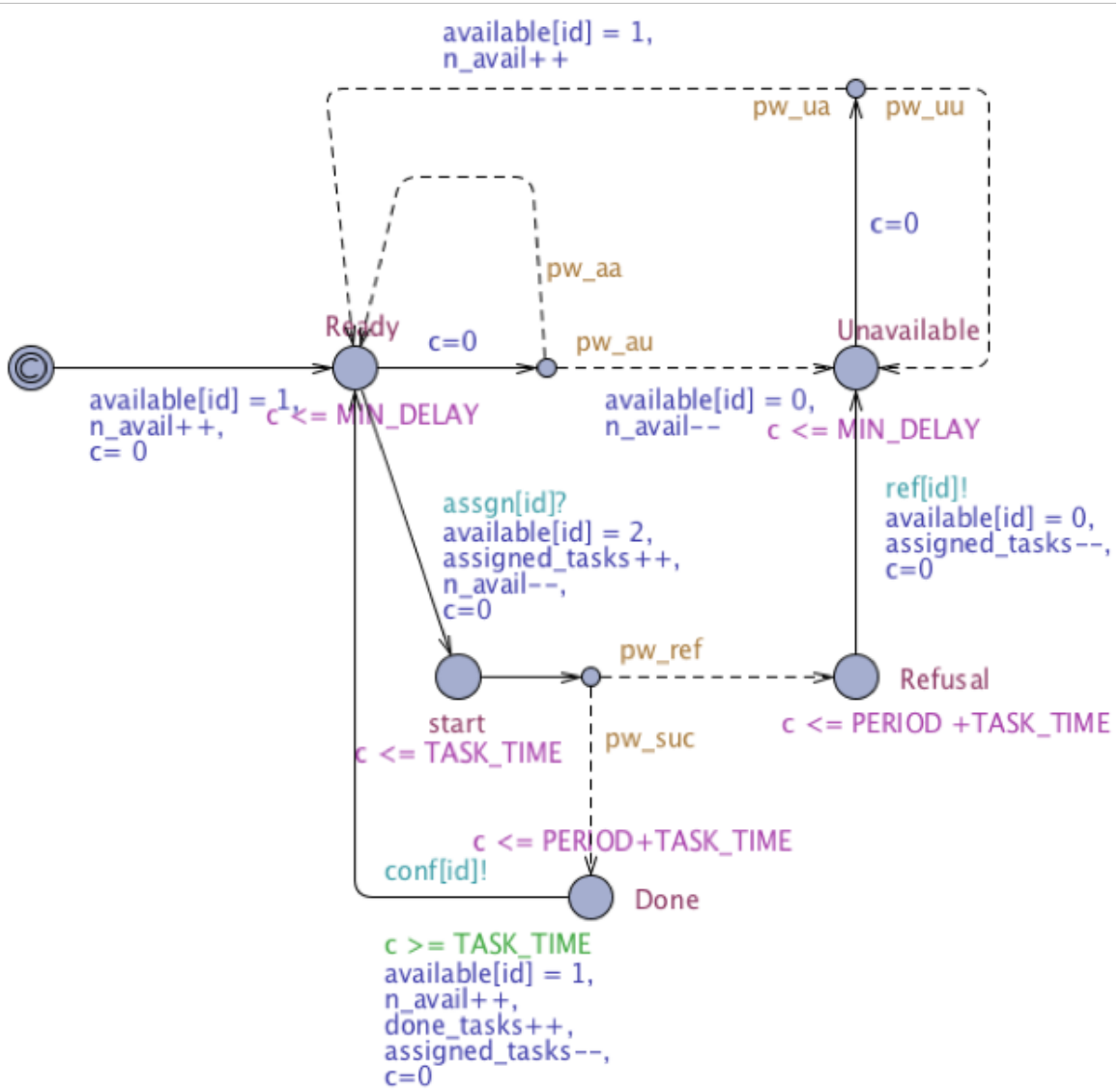
Formal models: examples

- 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 $\Rightarrow$ 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
- Succesful use of Event-B in the railway domain (e.g., verification of several driver-less metro lines in Paris)

Event-B

- 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 form 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.

Modelling in Event-B

- The dynamic system behaviour is described in terms of guarded commands (events):
  - Stimulus $\rightarrow$ 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

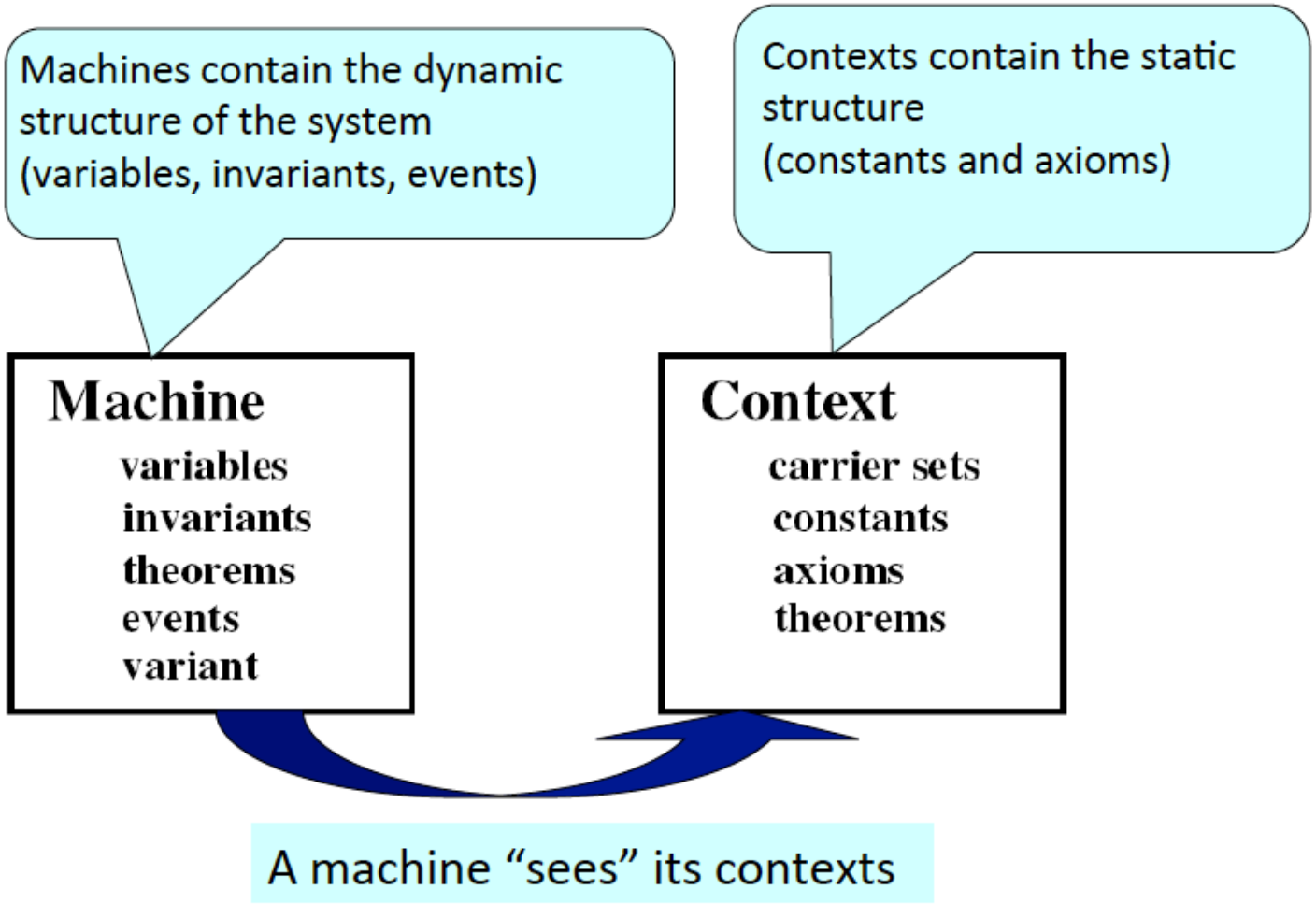
Event2 or

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



Rodin platform: automated environment for 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`