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.)

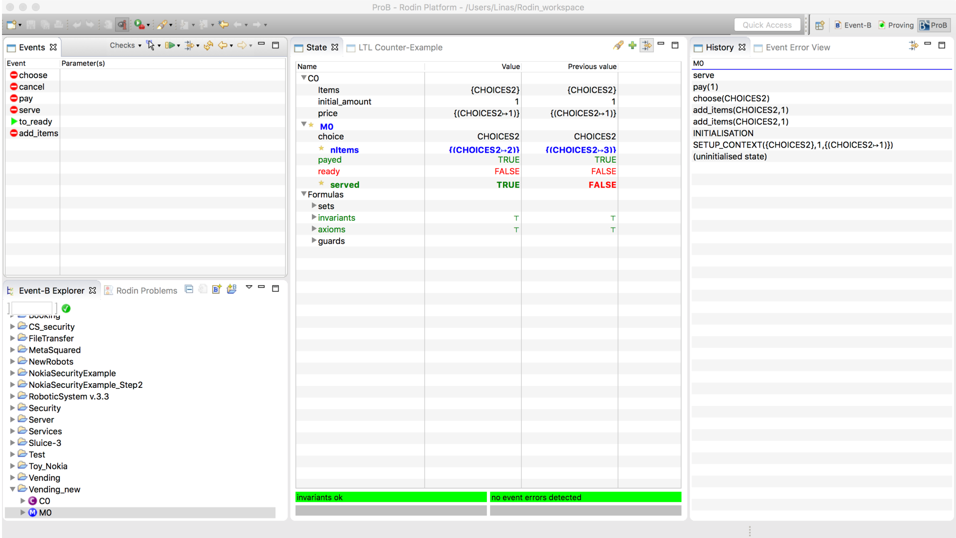
- Model checking usually relies on a generated *reachability graph* of system states
- A model checker explores all the state traces in the graph and checks the property (e.g. invariant preservation) in reached states
- Since a model checker operates on traces (not just single states), additional reachability 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

$$critical\_fault = TRUE \longrightarrow alarm = ON \wedge shutdown\_mode = TRUE$$

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.)



Event	Parameter(s)
choose	
cancel	
pay	
serve	
▶ to_ready	
add_items	

Name	Value	Previous value
▼C0		
Items	{CHOICES2}	{CHOICES2}
initial_amount	1	1
price	{(CHOICES2↦1)}	{(CHOICES2↦1)}
▼★ **M0**		
choice	CHOICES2	CHOICES2
★ **nItems**	**{(CHOICES2↦2)}**	**{(CHOICES2↦3)}**
payed	TRUE	TRUE
ready	FALSE	FALSE
★ **served**	**TRUE**	**FALSE**
▼Formulas		
▶sets		
▶invariants	T	T
▶axioms	T	T
▶guards		

invariants ok

no event errors detected

M0
serve
pay(1)
choose(CHOICES2)
add_items(CHOICES2,1)
add_items(CHOICES2,1)
INITIALISATION
SETUP_CONTEXT({CHOICES2},1,{(CHOICES2↦1)})
(uninitialised state)

- Booking
- CS_security
- FileTransfer
- MetaSquared
- NewRobots
- NokiaSecurityExample
- NokiaSecurityExample_Step2
- RoboticSystem v.3.3
- Security
- Server
- Services
- Sluice-3
- Test
- Toy_Nokia
- Vending
- Vending_new
  - C0
  - M0

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

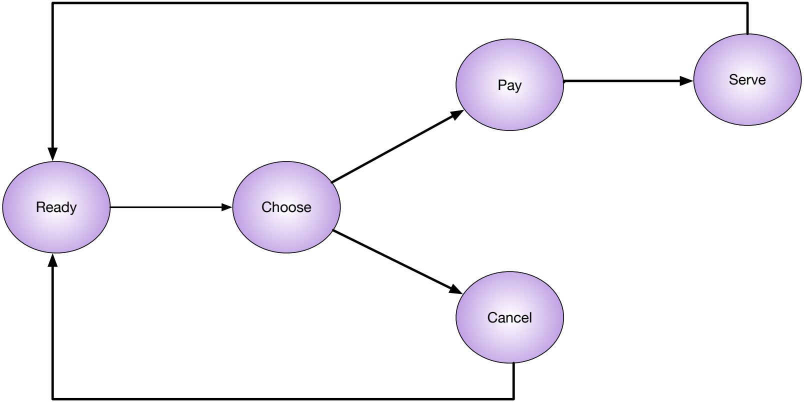
1. The vending machine serves the customer a product from a number of available choices;
2. The customer may select one of the available product choices;
3. After selection, the customer may proceed by paying for the selected product or cancelling the selection;
4. After sufficient payment, the selected 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(<formula>) – a formula holds all the time
    - F(<formula>) – a formula holds eventually
    - e(<event1,...>) – events enabled
    - {<condition>} – condition is true

- Example:
  G ({choice /= NONE} => F e(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