Towards the analysis of xUML models using an SMT solver

Osmar Marchi dos Santos
Jim Woodcock
Richard Paige

Department of Computer Science
The University of York
Summary

- The INESS Project
- Towards the use of an SMT solver
- xUML models of Interlockings
- Translation strategy
- Next steps
The INESS Project

- Integrated European Signalling System (INESS)
  - Integration of different signalling systems within Europe

- Led by the Union International des Chemins de Fer (UIC)
  - Composed of 30 different partners from Industry and Universities
The INESS Project

“With INESS, the trains will be able to cross all the borders in Europe without any trackside compatibility interference, something which has not been possible until now, particularly considering the various different national signalling systems”
The INESS Project

- Define a common core of properties for the different signalling systems within Europe and explore from a functional point-of-view

- Define models of this integrated system
  - Notion of (executable) specification
  - Defined in Executable UML (Artisan xUML tool)
  - Models can be analysed via simulation (Cassandra tool)
The INESS Project (Universities)

- Only simulation is available, Universities to provide strategies that enable the formal analysis of xUML models

- Universities include:
  - University of York
  - LaQuSo
  - Technical University of Eindhoven
  - University of Twente
  - University of Southampton
The INESS Project (Universities)

- Generate code (or specification) that is suitable for formal verification
  - Input to a formal verification analysis tool

- Different formal verification languages/tools
  - PROMELA (York)
  - μCRL2 (LaQuSo)
  - UML-B (Southampton)
The INESS Project (Universities)

- Formal verification strategy
  - Next talk in terms of $\mu$CRL2

Verification results in terms of xUML
(depends on how properties of interest can be formalized and represented at the correct level)
Towards the use of an SMT solver

- Initial translation to PROMELA (SPIN model checker) was defined
  - Published at FMCO last year

- Simple interlocking scenarios lead to state-explosion problems using model checking
  - Nevertheless, interesting verification results were obtained using μCRL2 (third INESS talk)
Towards the use of an SMT solver

- Scalability issues will arise
  - A large xUML model of an interlocking is composed of dozens of components

- Try to use new technologies for analysing concurrent software
  - Initial focus on automated theorem-proving (using an SMT solver)
Towards the use of an SMT solver

- Look at languages that could be used to automatically generate input to an SMT solver

- In particular, the use of Microsoft technology
  - State-of-the-art SMT solver (Z3)
  - SPEC# language being automatically translated
Towards the use of an SMT solver (SPEC# overview)

- Extends C# with contracts
  - Pre-conditions (*requires*)
  - Post-conditions (*ensures*)

- Definition of assertions over the program and loop invariants
Towards the use of an SMT solver (SPEC# overview)

- Provides a modular verification strategy
  - Only parts of the code that have been specified with pre/post-conditions are analysed
  - Analysis is generic

- Run-time checking is also supported
  - Like simulation, by executing the application
Towards the use of an SMT solver (SPEC# overview)

- The verification works by:
  - Translating code to Boogie (intermediate language)
  - Use Boogie to generate the Z3 model and analyse

- We would like to encode xUML in Boogie/Z3
xUML models of Interlockings

- Use of class diagrams and state machines
  - Every class diagram has an associated state machine
  - Inheritance is allowed

- State machines describe the behaviour of the associated class
xUML models of Interlockings (Class diagrams)

- Constructs used include, for instance:
  - Declaration of simple types, like integers
  - Declaration of derived attributes
  - Use of references

- Derived attributes allows to track the behaviour of different object states (defined by the state machine)
For example, use of keywords like “forall” and “exists” in the definition of derived attributes:

- Where the state(s) of one or more objects should be in a particular location (forall)
- Where the state(s) of at least one object should be in a particular location (exists)
xUML models of Interlockings (Class diagrams)

- Simple example of a (toy) Micro Interlocking:
xUML models of Interlockings (State machines)

- Transitions include:
  - Signal transitions
  - Time transitions (can be seen as non-deterministic choices, following the notion of time in the Cassandra simulator being used)
  - Change transitions (“when” expressions, that can use combinations together with derived attributes)
xUML models of Interlockings (State machines)

- States include:
  - Initial and simple
  - Composite (sequential and concurrent)

- States can include actions (*enter* and *exit*)

- Transitions as well
xUML models of Interlockings
(State machines)

- Since inheritance is supported, different state machines could be executing concurrently

- With respect to execution, a special “application” class (with associated state machine) is used
  - Actions are used to create the analysis scenario
  - There is no dynamic creation of objects
xUML models of Interlockings (State machines)
xUML Models of Interlockings (Behaviour)

Considering the transitions, the behaviour in the Cassandra simulator is defined as follows:

Whilst “change transitions” (“when” expressions) are enabled in the model, execute change transitions

If no “change transition” is enabled, execute “signal transitions” or “time transitions” (one at a time, since new “change transitions” may become enabled – due to the change of state in a different object)
Translation Strategy

- Initial work considered the generation of executable code in terms of SPEC#

- For instance, every UML class would become a SPEC# class

- Objects would be translated to threads and transitions would become methods (with pre and post conditions)
Translation Strategy

● For instance, a transition would be defined as follows (over variables defined in the class):

```java
public void transition_idle_to_preparing()
    requires route_state == STATE.idle;
    ensures route_state == STATE.active;
    ensures route_state_active == STATE.preparing;
{
    // Actions (send messages, change state)
}
```
Translation Strategy

- However, SPEC# does not take into consideration the execution of the threads for the verification

- Instead, run-time checks with simulation could be used
Translation Strategy

- In order to use the verification, we started focusing on building a simulation mechanism.

- Every object becomes a simulation object.

- Like in bounded verification, we want to be able to analyse the possible interactions within N steps (in a loop).
Translation Strategy

- Approach does not work, since loops in SPEC# need loop invariants

- And the way loops are translated to Boogie do not maintain the execution sequence
Translation Strategy

- Current approach we are looking at uses the notion of Context-Bound Analysis (CBA) (by Qadeer and Wu 2004; Lal and Reps 2009)

- Focuses on the idea of generating a sequential model that incorporates the concurrent behaviour of the model
Translation Strategy

- CBA has gained attention in recent years, due to its use with SMT solvers

- Analysis of the model is done within K steps, which also describe the possible context-switches (interactions) of the system
We are adopting the following translation strategy:

- Build a simulator, where each object becomes a component and states are defined globally
- The *random* type in SPEC# is used to introduce non-determinism for selecting context-switches
- The K execution steps are NOT implemented inside a loop construct
A structure like this follows:

```c
object_n = rnd.Next();
MESSAGENAME msg;
if (step < K) {
    if (object_n == route1) {
        if (state_space[route].route_state == STATE.idle) { // Do actions
            // Other transitions; other objects
```
Translation strategy

- Still working on how to deal with the priority given by the simulator regarding change transitions and how to encode efficiently the state space of the application
  - Because memory is shared, dealing with complex constructs, like “forall” and “exists”, found in the Interlocking models are easier to handle (compared to message passing)
Next steps

- Finish implementing the current proposed translation and generate example for the Micro Interlocking

- Look at other CBA approaches in the literature and how they can be implemented
  - Reducing the number of context-switches
  - Applying other optimisations
Next steps

- Compare results between:
  - Different CBA approaches
  - Against model checking

- Test scalability of the work

- Eventually provide a translation from xUML models directly in terms of Boogie or Z3
Towards the analysis of xUML models using an SMT solver

Osmar Marchi dos Santos
Jim Woodcock
Richard Paige

Department of Computer Science
The University of York