Towards Verification of Railway logic with Dafny and a Parameterized Model Checker

Gianluca Redondi, Trento

October 11, 2023

Contents

Symbolic Verification of Parameterized Systems

Symbolic Transition Systems

- Symbolic transition systems (over a background theory) are triples $S = (X, I(X), T(X, X'))$, where: (i) X is a set of variables; (ii) X' is a 'duplicate' of X, (iii) $I(X)$, $T(X, X')$ are formulae defining the initial states and the possible dynamic of the system.
- Formalism for modeling software, hardware, etc. . .
- A simple example with integer arithmetic:
  - $X = \{x, y\}$;
  - $I(X) \equiv x = 1 \wedge y > 0$
  - $T(X, X') \equiv x' = x + y \wedge y' = y$

  The model describes a simple system in which the value of x is increased at every step by a fixed value y.

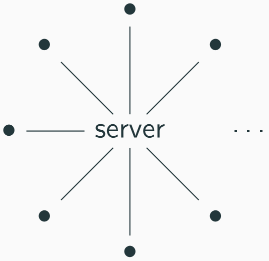
Given the system S defined by:

- $X = \{x, y\}$;
- $I(X) \equiv x = 1 \wedge y > 0$
- $T(X, X') \equiv x' = x + y \wedge y' = y$

The formula $x > 0$ is an invariant of the system. It is not, however, an inductive invariant.
An inductive invariant for the property is $x > 0 \wedge y > 0$, which can be automatically synthesized by e.g. SMT-based model checkers.

Parameterized systems

A parameterized systems is a systems in which the number of some components is left unbounded; e.g.: a client-server protocol with an unknown numbers of clients



Typical properties of parameterized systems (e.g. mutual exclusion) should hold independently by the numbers of components.

Formalization of Parameterized systems

- For the modelling of parameterized systems, quantifiers are needed in the formulae defining S;
- unbounded components can be modeled with a simple sort with equality (called the *index* sort);
- state variables are *functions* from the index sorts to some element sort.

For example, consider a protocol with two parameterized components: *tracks* (that can be locked or free) and *routes* (that can be active or inactive). A possible initial formula for the protocol could be

$$\forall t : \mathit{track}.(\mathit{state}[t] = \mathit{free}) \land \forall r : \mathit{route}.(\mathit{state}[r] = \mathit{inactive})$$

A railways example/2

An example of a transition formula:

$$\begin{aligned}&\exists r : \mathit{route}\big(\mathit{state}[r] = \mathit{inactive} \land \\ &\qquad \forall t1 : \mathit{track}(\mathit{Usedby}(t1, r) \rightarrow \mathit{state}[t1] = \mathit{free}) \land \\ &\land\, \mathit{state}'[r] = \mathit{active} \land \forall s : \mathit{route}(s \neq r \rightarrow \mathit{state}'[s] = \mathit{state}[s]) \\ &\qquad \forall t : \mathit{track}.\big(\mathit{Usedby}(t, r) \rightarrow \mathit{state}'[t] = \mathit{locked} \\ &\qquad\qquad \land \neg \mathit{Usedby}(t, r) \rightarrow \mathit{state}'[t] = \mathit{state}[t]\big)\end{aligned}$$

Suppose we define

$$\mathit{NotCompatible}(r1, r2) \iff r1 \neq r2 \wedge \exists t : \mathit{track}. \mathit{Usedby}(t, r1) \wedge \mathit{Usedby}(t, r2)$$

Then, a candidate property for the protocol can be:

$$\forall r1, r2 : \mathit{route}\big(\mathit{NotCompatible}(r1, r2) \rightarrow \neg(\mathit{state}[r1] = \mathit{active} \wedge \mathit{state}[r2] = \mathit{active})\big)$$

This is still not inductive; an inductive strengthening is

$$\forall r : \mathit{route}, t : \mathit{track}(\mathit{Usedby}(t, r) \wedge \mathit{state}[t] = \mathit{free}) \rightarrow (\mathit{state}[r] \neq \mathit{active})$$

Challenges

- In general, the invariant checking problem of symbolic transition systems (also without quantifiers) is undecidable;
- generation of quantified inductive invariant is scarcely supported;
- (un)satisfiability of first-order logic is an undecidable problem.

Yet...

- efficient algorithms for model checking systems without f.o. quantifiers;
- many methods to check the (un)satisfiability of f.o. formulae.

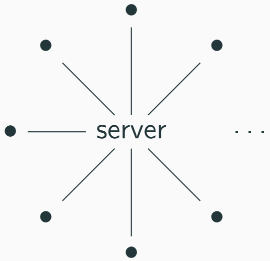
Automatic verification of Systems with quantifiers

We have developed two algorithms for the automatic discovery of (universally quantified) inductive invariants:

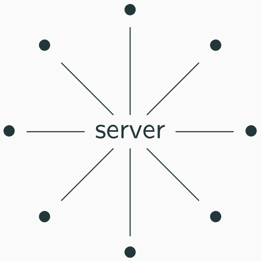
- A first algorithm is based on the following 'practical' intuition: *‘Either a counterexample occurs for a small value of the parameter, or the system is safe always for the same reason (at least after a certain threshold value)’*;
- A second algorithm is an extension of IC3 for quantified reasoning and theories: it has more theoretical properties, but in practice is inferior to the previous one.

A source of help: ground instances

From a parameterized systems (whose symbolic description requires quantifiers)...



..to a ground instance, where quantifiers are expanded in a finite set.



Ground instances are under-approximation of the parameterized system, but they are a useful source of heuristics, and can be model checked efficiently.

Generalization

- After a ground instance is checked, an inductive invariant for it is provided by the model checker;
- our procedure will try to lift such invariant to the parameterized (quantified) case;
- the validity of the new candidate invariant will then be checked by incomplete but efficient instantiation strategies.

AIDA: A tool for modeling, generating and verifying Safety Logic

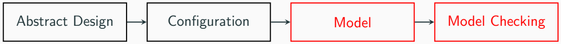
Railways Logic

- The Italian railways system is still *relay-based*. The safety of the control logic is ensured by the expertise of of railways engineer.
- There is a request for transfer to a computer-based system, where the safety can be also ensured by formal methods.
- Ideally, given a description of the logic, we would like to automate the proof of its safety.

Railways and AIDA

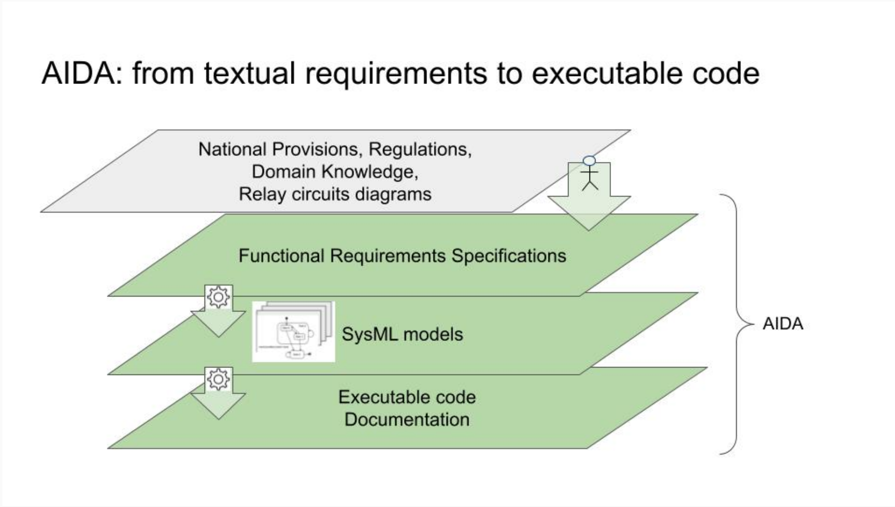
There is an ongoing collaboration between FBK and RFI; a tool (AIDA) for the future design of train stations:

- engineers write systems specs in pseudo-natural language: 'e.g. when a train is arriving, activate the correct switches';
- the design is then combined with a configuration: e.g. Trento's station has a certain number of tracks, some train uses a certain route, and so on;
- code is automatically generated for the functioning of every component of the station;
- 'standard' model checking can be now used to ensure safety-critical properties of the station.



automatic steps are in red

AIDA


AIDA: from textual requirements to executable code
National Provisions, Regulations, Domain Knowledge, Relay circuits diagrams
Functional Requirements Specifications
SysML models
Executable code Documentation
AIDA

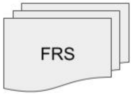
Structure of the Safety Logic (SL)

- The SL is organized in a set of classes
- Each class has a structure and a behavior
- Preserved at all levels: Textual FRS, SysML, code

- The structure may contain relations among class instances and parameters
  - E.g Itinerary has attribute endPoint: <s:Semaphore, tc:TrackCircuit, d:Direction>
  - Relations are instantiated at configuration time like the other parameters
- Class instances interact through the relations
  - By sending messages to other instances (*automatic commands*)
  - By reading parameters and variables of other instances
  - By writing variables of other instances
- Class instances interact with the environment
  - By receiving messages from the user (*manual commands*)
  - By reading inputs from the plant
  - By writing outputs to the plant

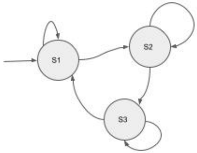
FRS at textual (controlled) natural language level

- Each class is specified in CNL in a *class sheet*
- Each class sheet:
  - Describe the *structure* of the class
    - Declare Parameters, Inputs, Outputs, Messages, State variables
    - Define Macros
  - Describes the *behaviour* of the class
    - Define states in *state sheets*
    - Each state sheet define a single state and all transitions originating from it

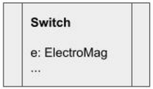


System

Class

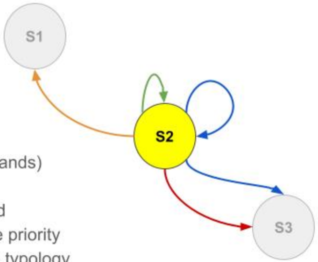


Behaviour



Structure

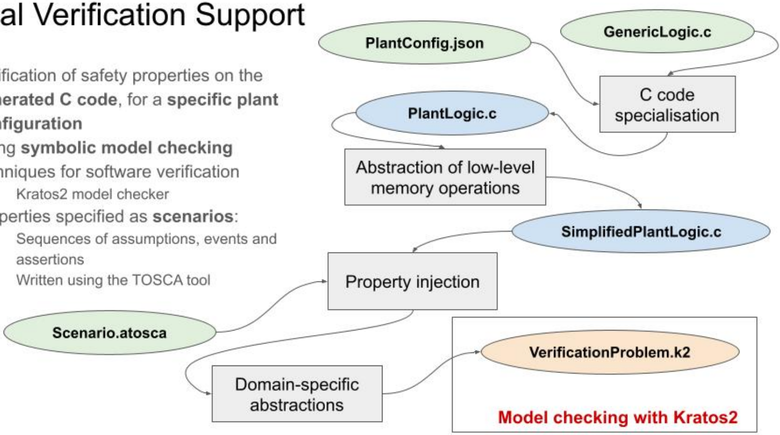
FRS: State transitions



- Each transition is characterized by:
    - Source state and destination state
    - An optional triggering event (manual or automatic commands)
    - A guard which enable the transition
    - An effect which is applied when the transition is executed
    - A typology (color in the figure) which helps determine the priority
    - Local priority to order evaluation of transitions with same typology
- Each transition is evaluated in a specific execution phase
    - *manual* (trigger is a manual command), *automatic* (automatic command) or *state* (no triggers)
    - (More later about *scheduling)*
- Global priority is determined from of phase, typology and local priority

Formal Verification Support

- Verification of safety properties on the **generated C code**, for a **specific plant configuration**
- Using **symbolic model checking** techniques for software verification
    - Kratos2 model checker
- Properties specified as **scenarios**:
    - Sequences of assumptions, events and assertions
    - Written using the TOSCA tool



Verification on generated code

The verification of the generated code can:

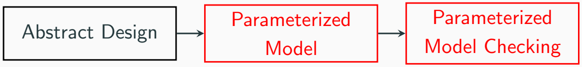
- Ensure that the code satisfies the specifications;
- Check the validity of some general properties (e.g. no trains collide);
- Generate counterexamples that are useful for testing.

However, this would check only that the particular configuration is correct or not.

Case study: RFI project /2

The design of the components is shared by all the different stations:

- Many properties should hold independently of the configuration!
- For example, the fact the two train never collide should be true independently of the number of lights, tracks, ...
- a parameterized model can be obtained much earlier in the verification process:



Our Goal

Given an abstract design of the railway logic, we would like to use to check automatically that:

i. the generated code satisfies the requirements independently from the configuration, and
ii. the design of the logic always satisfies some properties.

It is possible that for (i), invariants can be inferred automatically by patterns. For (ii) instead, the properties are dependent of the many interaction between the different components of the system.

Towards Dafny

Why Dafny?

It should be easy to generate Dafny code from the SysML models:

- Each class of the logic can be represented by a class in Dafny;
- transitions of the class are represented by Dafny methods;
- the body of such methods is filled with an abstract version of the generated code;
- FRS are translated as a set of pre- and post- condition;

Verification Dafny

- After the classes have been defined, we define a Station to be a set of class instances.
- To verify that a certain property hold, we check whether it is preserved by all methods of the classes.
- This may not be the case, as we may need a stronger (inductive) property...