Formal Verification for UML/SysML models

IBM Research Lab - Haifa
Content

- Formal verification v.s. testing
- Correctness properties
- Formal verification for Rhapsody models
Formal Verification

- Does the system obey its requirements?
- Demonstrate the cases where the system fails
- Exhaustive search for bugs
- In the absence of requirements
  - Try to violate universal rules that all systems should obey
From models to models

- A model of a system: UML / SysML
- Analysis model: finite state machine (mathematical)

Yes, these are huge! FV algorithms analyze them without building them
Verification amounts to analyzing computations looking for possible bad states

Imagine a system that reacts to its inputs and changes states

Assuming n Boolean variables:

**2^n** different states
Testing

- In testing we run the system on a single path through the computation tree
- Didn’t hit an error? We run again. And again. And again…
Formal Verification

- Formal verification uses **mathematics rather than chance**
Formal Verification

- Formal verification uses **mathematics rather than chance**
Formal Verification

- Formal verification uses **mathematics** rather than chance
Formal verification for UML/SysML

The user creates behavioral models

The user defines *correctness properties*

Fail 🙅

+ sequence diagram

Pass ☑️
Correctness properties

- First rule:  
  know what you want to verify!

- Second rule:  
  Say it clearly

- Formal verification uses *temporal specification languages*
  – Formally defined
  – Powerful
  – Not so easy to read / write

- Template properties hide the temporal language
  – Easy to understand, easy to use
  – Limited expressibility
Internal Non-Determinism

- Find scenarios in which there are two (or more) enabled transitions from the same state
- Model independent property

Can these two guards hold at the same time?

Non determinism

check for ND transitions
Out of Bounds

- Check that attributes cannot be assigned with out-of-bounds values
- Model independent property
Mutual Exclusion

- The user specifies two distinct states in two different objects
- The tool verifies that these states can never be active at the same time
- Model specific property
An invariant is an expression that should hold at all times.

Invariants

- Invariant expressions can refer to states and attributes
  - `a1` is an object name
  - `state(a1)` is the current state of `a1` (can be hierarchical)
  - `a1.x` is the attribute `x` of `a1`

- Model specific property
Dead Code

- Find *states* that cannot be reached
- Find *transitions* that cannot fire
- Model independent property

Dead States

check for dead states
Deadlock Freedom

- A deadlock is a situation in which no progress can be made regardless of what the environment does.
  - In other words, no transitions can ever be fired
- Model independent property
… and more

- Template properties can be added and customized on-demand
  - Should be tailored to in-house design methodology of specific customer

- Verification methodology:
  - Template properties used by engineers
  - Temporal logic used by verification expert

- Correctness properties should be part of the model
The Model Verifier

- Verifying UML/SysML (behavioral) specifications
  - A subset defined for safety critical systems
Contact information

- Ronen Levy
  Emerging Quality Technologies, Manager
  E-mail: ronenl@il.ibm.com

- Karen Yorav
  FV for UML project leader
  E-mail: yorav@il.ibm.com