Object Oriented Modeling using Alloy

Mark Reynolds

markreyn@cs.bu.edu

Outline

- Goals
- Modeling
- The Alloy modeling language
- Alloy for static modeling
- Alloy for dynamic modeling
- Alloy course projects

Goals

- Given a software system, you want to analyze its "properties"
- For example, for a block of code you might be interested in the following properties:
  - Global/Local variable utilization
  - Stack/Heap/Memory utilization
  - External resource utilization (files, sockets, …)
  - Security properties
- How?
  - Source code analysis tools
  - Binary code analysis tools
  - Debuggers
  - Tracing
  - Logging

Modeling

- Build a model of the code
  - Invariants/Constraints (static properties of code)
    - Length of each instruction
    - Stack properties of each instruction
    - Registers used by each instruction
  - Assertions (how the code "should" work)
    - "The stack depth should never be negative"
    - "A register must be written before it is read"
    - "Memory locations [X:X+128] are read-only"
- Run the model
  - Are the assertions actually satisfied?

Synthetic Execution

- Incorporate state into the model
- Add state transition rules
  - When executing a “push” instruction add 1 to the stack depth
- Initialize the model state corresponding to the machine state at the beginning of the block of code being analyzed
- Run the model and check the assertions at each reachable state
  - Satisfying instances => assertions true
  - Counterexamples => assertions false

Execution Tracing

- Actually run the code
- Gather trace/log data
- Combine the data with the assertions to create a complete model
- Run the model
  - Satisfying instances => assertions true
  - Counterexamples => assertions false
- Execution tracing only checks the assertions for the states that are reached, while synthetic execution checks the assertions that could be reached

Advantages of the model-based approach

- Can ask very specific questions
- Can check specification versus implementation
- Can be developed in a modular, incremental fashion
- More general than a tools-based approach

Alloy

- Based on sets and relations (mappings) between them
- Simple declarative syntax
- Supports complex data structures
  - Provides an object oriented programming paradigm
  - Well suited to describing state
- The Alloy analyzer is sound
  - Never produces false positives
- The Alloy analyzer is complete up to scope
  - Never produces false negatives within a given scope (set size)
  - Based on the “small scope hypothesis”
- alloy.mit.edu

Alloy for static modeling

- Filesystem model (from the Alloy tutorial)
  - Objects: files and directories
  - Invariants:
    - An object is either file or a directory
    - There is a single root directory
    - Every object is reachable from the root
  - Relations:
    - Parent: maps an object to its parent directory
    - Content: maps a directory to the objects it contains

Filesystem Model

sig FSObject { parent: lone Dir }

sig Dir extends FSObject
{ contents: set FSObject }

sig File extends FSObject { }

Alloy syntax

- A *sig* or *signature* defines a set
- The keyword *extends* is used to define a subset
- Relations are defined within the signature body
  - The domain of the relation is the signature/set
  - The range of the relation is given after the colon
  - Relations can have quantifiers: *one*, *lone*, *set*, …

Filesystem Model

sig FSObject { parent: lone Dir }

“FSObject is a set of objects; it has a relation ‘parent’ which is a map between that object and at most one directory”

sig Dir extends FSObject
{ contents: set FSObject }

“Dir is a subset of FSObject; it has a relation ‘contents’ which is a map between that directory and a set of filesystem objects”

sig File extends FSObject { }

“File is a subset of FSObject”

Filesystem Invariants

fact { File + Dir = FSObject }
*fact { FSObject in Root.*contents }*
fact { all d: Dir, o: d.contents | o.parent = d }

fact introduces a standalone invariant/constraint
A *fact* must always be true in the model
+ is set union
in is set membership
| is “satisfied”
^ is transitive closure
    d.^contents = d.contents + d.contents.contents + …
* is reflexive transitive closure
    d.*contents = d + d.^contents

Filesystem Invariants

one sig Root extends Dir { } { no parent }

Combination of a set declaration and an invariant/constraint:

1. There is a object Root that is a subset of Dir
2. There is exactly one such object (*one* quantifier)
3. When the *parent* mapping is applied, the range is empty (*no* quantifier)

Same as

one sig Root extends Dir { }

fact { no Root.parent }

Filesystem Invariants

fact { File + Dir = FSObject }

“Every filesystem object is either a File object or a Dir object”

*fact { FSObject in Root.*contents }*

“Every filesystem object is part of the set of objects consisting of the Root object itself and the set obtained by applying (repeatedly) the ‘contents’ relation”

fact { all d: Dir, o: d.contents | o.parent = d }

“A directory is the parent of its contents”

Alloy Assertions

- Facts are what must be true, assertions are what you are checking
- Assertions are always checked within a given scope
  - The scope is the maximum size of the set of objects that will be checked
  - The "small scope hypothesis" says that interesting counterexamples will almost always be found with small scope values
  - Alloy is not a proof system

Filesystem assertions

- “No directory ever contains itself (the filesystem has no cycles)”

assert acyclic { no d: Dir | d in d.^contents }
check acyclic for 5

assert keyword introduces an assertion
check causes the analyzer to run
for keyword specifies the scope
In this case, all models with not more than 5 FSObjects will be checked

The Filesystem Model

```
sig FSObject { parent: lone Dir }
sig Dir  extends FSObject { contents: set FSObject }
sig File extends FSObject { }
one sig Root extends Dir { }

fact { File + Dir = FSObject }
fact { FSObject in Root.*contents }
fact { all d: Dir, o: d.contents | o.parent = d }
fact { no Root.parent }

assert acyclic { no d: Dir | d in d.^contents }
check  acyclic for 5
```

Alloy for dynamic modeling

- Create a model of a state machine
- Use Alloy’s *ordering* utility to define relations *first*, *last* and *next* for States
- Derive the initial state of the state machine from the static properties of the system
- Specify the state transition rules
- Write the assertions you want to check for each State
- Run the model and test the assertions
- “River Crossing Model” in the Alloy tutorial

CS511 Alloy Projects

- Based on the Java Virtual Machine (JVM)
- Given a Java method, create a static/dynamic model of certain properties of the binary code
- Write assertions to be checked in Alloy
- Run the analyzer and verify/refute your assertions

Contact Information

- Email: markreyn@bu.edu
- Office: MCS 181
- Office hours: by appointment
- JVM mini-tutorial
  - Mon, Jan 31, 5:30-6:30 PM