Test-Generation Tools

Derek Rayside
drayside@swen.uwaterloo.ca

March 29, 2010

1 When to use a Test-Generation Tool

A thorough test plan should involve both manually written tests and a machine-generated tests. You should use a test-generation tool whenever the technology is available.

Test-generation tools are most mature for general properties that all software should have, such as avoiding deadlock, object contract compliance, closing files that have been opened, avoiding segmentation faults, avoiding null-pointer dereferences, *etc*. Tools that test these general properties are also usually the easiest to use because they do not require writing a program-specific specification — the specification of the general property comes baked into the tool.

There are also tools for checking program-specific properties. These tools tend to focus on a particular kind of property. For example, tools like Spin and SMV are targetted to concurrency properties. If you are testing a program that involves concurrency or data structures you should definitely be using a test-generation tool.

Test-generation technology is just starting to enter the mainstream. For example, Microsoft now requires Windows device drivers to pass the Static Driver Verifier. Most of the tools discussed in this lecture have been developed in the research community. However, all are available for download and use. Some, such as Spin, have been in heavy use for over 15 years.

2 Testing Programs & Testing Ideas

Some test-generation tools, such as Randoop, are intended to test real production code. Other tools, such as JPF, are intended to test reduced programs that capture the essence of the computation. Finally, some tools, such as Alloy, Spin, and SMV, are intended to test ideas rather than implementations.

Why would you ever want to test an idea? A program that implements a broken idea is never going to really work, so the first order of business is to test that the idea makes sense. Any program involving concurrency, parallelism, or distributed computation falls into this class. Consider, for example, the famous Dining Philosophers problem. Many broken solutions have been proposed for this problem over the years. The only way to really know if a potential solution actually works is to use an appropriate test-generation tool to check the idea first.

3 Specification Languages

Using a test-generation tool for general properties, such as object-contract compliance, usually does not require writing any program-specific specification, since the general property is already baked into the test-generation tool.

Testing program-specific properties requires writing a specification for the program under test. For example, one might want to express the property that a linked-list should be acyclic.

Some tools, such as Korat, expect program-specific specifications to be written in the programming language (*e.g.*, Java). This makes it easy to write the specification because one does not have to learn a new language. However, sometimes it is difficult to write the specification one wants to write. For example, acyclicity of linked structures is more directly and concisely expressed in Alloy than in Java.

Specification languages for program-specific properties are often extensions of first-order logic (FOL): *i.e.*, first-order logic plus feature X. Here is a brief summary of the specification languages discussed in this lecture:

Propositional Logic. Recall (from some first-year course) that propositional logic involves only boolean variables and simple connectives such as $\wedge$ (and), $\vee$ (or), $\neg$ (not), *etc.* Propositional logic is what a SAT solver works with. Recall (from some theory of computation course) that the SATisfiability problem is to find an assignment of truth-values to the variables in a propositional forumula such that the formula evaluates to true.

First-Order Logic (FOL). Recall (from the same first-year course) that first-order logic adds the quantifiers $\forall$ (all) and $\exists$ (some/exists) to propositional logic. First-order logic is strictly more expressive than propositional logic.

Alloy: FOL + Relations + Transitive Closure. The Alloy logic language is a first-order logic extended with relations (*i.e.*, tables) and transitive closure (the * operator). Relations are a convenient way to model pointers/references in the program heap. Consider an object x such that $x.\mathsf{next} = y$ (the `next` field of x refers to object y). We could model this field reference with a tuple (*i.e.*, row) $\langle x, y\rangle$ in a relation (*i.e.*, table) called 'next'. Relational logic is the conceptual foundation of relational databases.

The *transitive closure* of a relation (table) tells us where we can get to from here. Suppose the 'next' relation has the tuples $\langle x, y\rangle$ and $\langle y, z\rangle$. The transitive closure of the 'next' relation would also include the tuple $\langle x, z\rangle$, because we can get to object z from object x by following the `next` field twice. The SQL'99 standard includes a transitive closure operator, but few database engines implement it.

The combination of relations and transitive closure make Alloy very convenient for specifying linked data structures. For example, it is very easy to say 'this list is acyclic': we just say that no node in the list can reach itself by following the `next` field.

Linear Temporal Logic (LTL): FOL + time. LTL adds a concept of time to first-order logic, as well as the temporal quantifiers 'eventually' ($\Diamond$), 'always' ($\Box$), and 'holds in the next state' ($\bigcirc$). LTL also includes binary operators for 'until' and 'release' but we did not discuss them in this lecture.

Computation Tree Logic (CTL): FOL + time. CTL is another logic for talking about time. Whereas LTL views time as a line, CTL views time as tree-like. We did not discuss CTL during the lecture. The important thing for your to remember is that there are some kinds of temporal properties that are only expressible in LTL, and others that are only expressible in CTL — and some properties that can be expressed in either. We did not discuss which properties each logic is better suited for. Some newer model-checking tools support both CTL and LTL.

4 Finitization

Traditional engineering uses both mathematics and approximations. From the perspective of this lecture software engineering, and particularly software testing, are no different. We use discrete mathematics (*e.g.*, logic) rather than the continuous mathematics (*e.g.*, calculus) of physics. The kinds of approximations we make are also different. The kind of approximation made by software testing tools is most often some kind of *finitization*.

Hardware systems are finite. There are only so many pins, so many transistors, so many states. This is very convenient from a testing point of view because we can, in principle, test every possible input and every possible state.

Software systems, by contrast, are usually infinite. The term 'infinite' is a bit of a misnomer because software always runs on finite hardware: in the terms of your theory of computation class, we're never actually running Turing Machines (which have infinite tape), but rather Linear Bounded Automata (which have finite tape). In any event, most programs are often considered 'infinite state systems'. What this really means is that we do not know at compile/testing time how large a given linked-list is going to be for some unknown future input.

In an infinite state system we cannot, in finite time, test every possible state nor every possible input. So what we do instead is some kind of *finitization* approximation: *i.e.*, we somehow make a finite-state system that approximates our infinite-state system of interest, and we test that finite-state system instead. There are three main ways in which this kind of approximation can be done:

Manual Finitization. The tester manually constructs a finite-state system that they believe is a reasonable approximation of the system of interest. Spin and SMV are such tools: they test finite-state machines. If you wish to check an infinite-state system with these tools then you must manually construct an approximating finite-state system.

Search Time. The tool attempts to examine every possible state/input. Since there are infinitely many possible inputs/states, this process will never terminate naturally. The tester simply terminates the tool after some period of time, and whatever was checked in that time is the finite approximation.

Randoop is a tool that uses this finitization strategy. Randoop performs a randomized search, so that any possible state is equally likely to have been checked. A systematic search, by contrast, will only check the beginning of the space.

Bounding. Putting a finite bound on something that is, in principle, unbounded. For example, ESC/Java unrolls all loops once — essentially it converts loops in the control flow graph to conditionals (`if` statements).

Alloy, Forge, and Korat all have the user/tester put bounds on the number of objects to be considered (*i.e.*, the heap). For example, Korat will generate all non-isomorphic binary trees up to length 4 (or whatever bound the tester chooses). The conjecture that exhaustively examining all small examples up to some finite bound is a reasonable finite approximation of the infinite-state system under consideration is known as *the small-scope hypothesis*. The small-scope hypothesis is generally believed to hold for data-structures and other kinds of systems that have an inductive structure (a base case followed by an incremental case).

5 Testing Tools

These are all tools that you can download and use today. There are many other testing tools available. This is a small selection of some tools that I think are useful and/or important. There are other important and useful testing tools that we have not discussed in this lecture. A brief description of the tools follows the summary table.

Tool	**ESC/Java**	**Forge**	**Alloy**	**Korat**	**Randoop**	**JPF**	**Spin**	**SMV**
Primary Use	Data Structures				Object Contract	Concurrency		
Specification Language	JML	Alloy		Java			LTL	CTL
Implementation Language	Java		Alloy	Java			Promela	?
Approximation	loop unrolling	heap size			search time		finite input	
Technology	Theorem Prover	SAT Solver		custom search				BDD
Author	Compaq	MIT		UIUC	MIT	NASA		CMU

ESC/Java. ESC/Java checks Java programs against specification written in JML (Java Modelling Language). This was covered in a tutorial earlier in the term.

Forge. Forge is similar to ESC/Java from the user's perspective, except it uses an Alloy-like specification language instead of JML. The behind-the-scenes technology is also different. One important difference from the user's perspective is that Forge gives concrete counter-examples if the program does not meet its specification: *i.e.*, Forge tells you which test-input leads to the bad state.

Alloy. The Alloy Analyzer tool translates the Alloy logic language (discussed above) to SAT (also discussed above). Use this to check that two logical statements are equivalent to each other. Perhaps one logical statement describes what the end result of adding an element to a list is, and another statement describes how to add an element to a list: these two statements should be logically equivalent.

Korat. Korat generates Java objects from a representation invariant specification that is written as a Java method. Consider a `BinaryTree` Java class. The `repOk()` method will return true if the tree is well-formed. For example, it might check for each node in the tree that the left and right pointers do not refer to the same child node. Given this `repOk()` method, Korat will generate all non-isomorphic trees up to a given size (say, 4). These trees can then be used as inputs for testing the `add()` method of the tree class (or any other methods in the class).

Randoop. Randoop generates random sequences of method calls looking for object contract violations. Just point it at the program under test let it run. Randoop discards method sequences that result in illegal argument exceptions, *etc.* It remembers method sequences that construct complex objects, and sequences that result in object contract violations.

Java Path Finder (JPF). JPF is a special Java Virtual Machine that, by default, explores different thread interleavings looking for concurrency bugs.

Spin. Spin checks finite-state machines written in Promela against specifications written in LTL. Most commonly there are a set of finite-state machines. For example, in the Dining Philosophers problem each philosopher would be represented by an independent finite-state machine. Spin is generally used to look for overall system problems such as deadlock, fairness, *etc.*

SMV. SMV is, at a high-level, similar to Spin. As discussed above, SMV checks finite-state machines against specifications written in CTL, which can express some properties that LTL cannot express.

A Homework

1. Draw a decision-tree to help a software tester select an appropriate tool (if any) for a given program. More than one tool may be appropriate for a given program. For example, ESC/Java and Forge are applicable in essentially the same set of circumstances.

2. Give an example of how you could apply one of the three finitization strategies to a program that you tested this term.