Course Document for
Analysis of Software Artifacts

Somesh Jha
Computer Science Department
University of Wisconsin
Madison, WI 53703.
email: jha@cs.wisc.edu

1 Course Overview

Software engineering process goes through several phases, e.g., requirements, design, prototyping, implementation, and testing. Artifacts such as message sequence charts, state machines, code, and test plans are produced during each phase of the process. The purpose of this course is to provide analysis techniques for analyzing various artifacts produced during this process. The course will concentrate on techniques that are formal and can be automated.

2 Schedule

2.1 Analysis of State Machines

State machines are useful for modeling distributed and concurrent systems. Statecharts [4] a formalism which is a part of UML [1] are essentially state machines. *Model checking* is a technique for verifying properties of state machines. Initially model checking was used for verifying hardware designs. Recently researchers have started investigating applications of model checking to software [2, 3].
References:

1. *Model checking*, E.M. Clarke, O. Grumberg, and D.A. Peled, MIT Press, 2000.

2. *The Model Checker Spin*, G.J. Holzmann, IEEE Transactions on Software Engineering (TSE), Vol 23/5, pages 279-295, May 1997.

3. *Model Checking for Programming Languages using VeriSoft*, P. Godefroid, Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL), pages 174-186, Paris, January 1997.

Tools:

Lecture	Date	Day	Subject
1	9/6	W	Course overview
2	9/8	F	Course overview
3	9/11	M	Model checking basics
4	9/13	W	Model checking basics
5	9/15	F	Model checking basics
6	9/18	M	Model checking basics
7	9/20	W	Introduce SMV
8	9/22	F	Introduce SMV
9	9/25	M	Other model checkers
10	10/2	M	Other model checkers
11	10/4	W	Model checking real-time systems
12	10/6	F	Model checking real-time systems

Figure 1: Schedule for analysis of state machines

1. *SMV:* A model checker built first at Carnegie Mellon University. However, we will be using the reengineered SMV called *NuSMV* developed at ITC-IRST.
   website: `http://sra.itc.it/tools/nusmv/`.

2. *Spin* is a widely distributed software package that supports the formal verification of distributed systems. The software was developed at Bell Labs in the formal methods and verification group starting in 1980.
   website: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

3. *VeriSoft is* a tool for software developers and testers of concurrent/reactive/real-time systems.
   website: `http://www1.bell-labs.com/project/verisoft/`.

Schedule: See Figure 1.
Note: There will be no class on 9/27 and 9/29. I have to go out of town for an important business.

2.2 Analysis of Abstract Data Structures (ADTs)

Formalisms based on state machines are appropriate for modeling concurrent and distributed systems. However, these formalisms are not very suitable for expressing abstract data structures (ADTs), e.g., modeling an abstract stack and all the associated operations. Formal languages that are based on relations are particularly appropriate for this purpose. We will cover such a formal language called *nitpick* or *NP* for short. A tool *ladybug* for checking properties of designs expressed in NP will also be explored. *Alloy* is a relational language for expressing object oriented designs. Alloy is inspired by UML. We will also explore the *Alloy Constraint Analyzer*, a tool for verifying designs expressed in Alloy.

References:

Lecture	Date	Day	Subject
13	10/9	M	Nitpick
14	10/11	W	Nitpick
15	10/13	F	Ladybug
16	10/16	M	Alloy
17	10/18	W	Alloy
19	10/20	F	Alcoa

Figure 2: Schedule for analysis of ADTs

1. *Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector.* , Daniel Jackson and Craig Damon, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), San Diego, CA, January 1996.

2. *Alloy: A Lightweight Object Modeling Notation*, Daniel Jackson July 2000.
   website: `http://sdg.lcs.mit.edu/~dnj/publications.html#alloy`

Tools:

1. *Ladybug* is a tool for analyzing designs written in NP.
   website: `http://www.cs.cmu.edu/~nitpick/ladybug.html`

2. *Alloy* is an object modelling notation that is compatible with development approaches such as UML, Catalysis, Fusion, OMT and Syntropy. It has a simple ASCII textual notation of which a subset can be expressed graphically. The *Alloy Constraint Analyzer* is a tool for analyzing Alloy models. It offers fully automatic animation: it can generate states that satisfy invariants, executions of operations, and counterexamples to claimed properties of a model.
   website: `http://sdg.lcs.mit.edu/alloy/`

Schedule: See Figure 2.

2.3 Testing

Assumed that you have finished the requirements and design phase. Once you start coding, the *testing phase* starts. There are two major forms of testing, *black box* and *white box* testing. In black box testing the tester ignores the internals of the program (or the tester might not have access to the code) and simply compares the output of the program being tested to the specification. In *white box* testing the tester inspects the code and designs the tests accordingly. *Testing metrics* are a measure of how thorough the test suite is. In this course we will primarily cover white box testing and testing metrics.

References:

1. *Selecting software test data using data flow information*, S. Rapps and E. Weyuker, IEEE Transactions on Software Engineering (TSE), Vol SE-11, April 1985, pages 367-375.

Lecture	Date	Day	Subject
20	10/23	M	Data Flow Analysis
21	10/25	W	Data Flow Analysis
22	10/27	F	Testing Metrics
23	10/30	M	Testing Metrics
24	11/1	W	Markov chains
25	11/3	F	Markov chains
26	11/6	M	Usage based testing
27	11/8	W	Usage based testing

Figure 3: Schedule for testing

2. *Markov Analysis of Software Specifications*, J. A. Whittaker and J.H. Poore, ACM Transactions. on Software Engineering Methodology (TOSEM), Vol. 2 No. 1, January, 1993, pp. 93-106.

3. *The Art of Software Testing*, G.J. Myers, John Wiley and Sons, 1979.

Tools: None. **Schedule:** See Figure 3.

2.4 Program Understanding

Until now we have assumed the program is being constructed from scratch. Unfortunately, most programs use (or have to use) "legacy code" or existing code for the project. Therefore, frequently one has to understand legacy code and construct a model of it. This process is usually called *reverse engineering*. In this part of the course we will concentrate on tools for program understanding, i.e., the process of understanding existing code.

References:

1. *Ciao: A Graphical Navigator for Software and Document Repositories*, Y.R. Chen, G.S. Fowler, E. Koutsofios, R.S. Wallach.

2. *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, M.D. Ernst, J. Cockrelly, W.G. Griswold, D. Notkin, *submitted to* IEEE Transactions of Software Engineering (TSE).

Tools:

1. CIAO is a customizable and extensible navigator. It allows users to query, analyze, visualize, and track structures of various software and document repositories. CIAO has been instantiated for C, C++, HTML, Java, Ksh documents, and several business repositories.
   website: `http://www.research.att.com/∼ciao/`.

2. *Daikon* (named after the Asian radish) is a prototype implementation of invariant detection. Front ends exist for C/C++, Java, and Lisp. Daikon is a research prototype and not rock-solid, nor is it intended for commercial use.
   website: `http://sdg.lcs.mit.edu/∼mernst/daikon/`.

Lecture	Date	Day	Subject
28	11/10	F	Call graphs
29	11/13	M	Extracting invariants
30	11/15	W	Extracting invariants

Figure 4: Schedule for testing

Lecture	Date	Day	Subject
31	11/17	F	Safety checking of machine code ($\star$) Z. Xu will deliver lecture 31.
32	11/20	M	Secure Information Flow
33	11/22	W	Secure Information Flow

Figure 5: Schedule for Code Safety.

Schedule: See Figure 4.

2.5 Code Safety

Since the advent of the *World Wide Web* (henceforth referred to as *WWW*), foreign and untrusted code is routinely executed on one's computer. Therefore, code safety has become a very important issue. This part of the course will focus on analyzing code to ensure safety.

References:

1. *Safety Checking of Machine Code*, Z. Xu, B. Miller, and T. Reps, proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2000.

2. *Protecting Privacy using the Decentralized Label Model*, A.C. Myers and B. Liskov, *to appear in* ACM Transactions on Software Engineering Methodology (TOSEM).

Tools: None.
Schedule: See Figure 5

2.6 Closing and Presentations

We will devote two lectures to closing and comments. The rest of the lectures will be devoted to student presentations.

References: Not applicable.
Tools: Not applicable.
Schedule: See Figure 6.
Note: 11/23-11/26 is Thanksgiving break.

Lecture	Date	Day	Subject
34	11/27	M	Closing comments
35	11/29	W	Comments session

Figure 6: Schedule for Closing

2.7 Grading in the course

There will be *no exams* in the course. Regular homeworks will given in class. Homeworks will account for 50 percent of the class. Students will also complete a project in the class. Please see the project guidelines on the class homepage. Project will account for 50 percent of the class.

References

[1] G. Booch, Ivar Jacobson, James Rumbaugh, and Jim Rumbaugh. *The Unified Modeling Language User Guide (The Addison-Wesley Object Technology Series)*. Addison-Wesley, October 1998.

[2] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[3] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera : Extracting finite-state models from java source code. In *in Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[4] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw Hill, October 1998.