Course Document for
Analysis of Software Artifacts

Somesh Jha
Computer Sciences Department
University of Wisconsin
Madison, WI 53706.
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 techniques for analyzing various artifacts produced during the software process. The course will concentrate on techniques that are formal and can be automated.

Following books are required for the course. All other books mentioned in this document are optional. Copies of papers mentioned in this document will be handed out it in class or a link to a printable version of the paper will be provided.

- *Model checking*, E.M. Clarke, O. Grumberg, and D.A. Peled, MIT Press, 2000.
- *The Unified Modeling Language User Guide (The Addison-Wesley Object Technology Series)*, G. Booch and Ivar Jacobson and James Rumbaugh and Jim Rumbaugh, Addison-Wesley, October, 1998.

2 Schedule

2.1 Analysis based on State Machines

State machines are useful for modeling distributed and concurrent systems. *Statecharts* [HP98] a formalism which is a part of UML [BJRR98] 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 [CAB+98, CDH+00, DHJ+00, BR01, BMMR01].

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), 23(5), 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.

4. *Tool-supported Program Abstraction for Finite-state Verification*, Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Willem Visser and Hongjun Zheng, Proceedings of the 23rd International Conference on Software Engineering (ICSE), May, 2001.

5. *Automatic Predicate Abstraction of C Programs*, T. Ball, R. Majumdar, T. Millstein and S. K. Rajamani, PLDI 2001, ACM SIGPLAN Notices.

Tools:

1. *SMV:* A model checker first implemented at the Carnegie Mellon University. However, we will be using the reengineered SMV called *NuSMV* developed at ITC-IRST, Italy.
   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/`.

4. The goal of the Bandera project is to integrate existing programming language processing techniques with newly developed techniques to provide automated support for the extraction of safe, compact, finite-state models that are suitable for verification from Java source code. While our ultimate goal is fully-automated model extraction for a broad class of software systems, our approach takes as a given that guidance from a software analyst may be required.
   website: `http://www.cis.ksu.edu/santos/bandera/`

5. The SLAM project at Microsoft Research investigates the relationships between Software (Specifications), Languages, Analysis, and Model checking. Our goal is to be able to check that software satisfies critical behavioral properties of the interfaces it uses, and to aid software engineers in designing interfaces and software to ensure reliable and correct functioning. The SLAM project is an effort of the *Software Productivity Tools Research group.*
   website: `http://www.research.microsoft.com/slam/`.

Schedule: See Figure 1.

Lecture	Date	Day	Subject
1	9/5	W	Course overview
2	9/7	F	Project overview
3	9/10	M	Model checking basics
4	9/12	W	Model checking basics
5	9/14	F	Model checking basics
6	9/17	M	Model checking basics
7	9/19	W	Introduce SMV
8	9/21	F	Introduce SMV
9	9/24	M	Other model checkers
10	9/26	W	Other model checkers
11	9/28	F	Other model checkers
12	10/1	M	Other model checkers
13	10/3	W	Model checking real-time systems
14	10/5	F	Model checking real-time systems

Figure 1: Schedule for analysis of state machines

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 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:

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 modeling 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

Lecture	Date	Day	Subject
15	10/8	M	Nitpick
16	10/10	W	Nitpick
17	10/12	F	Ladybug
18	10/15	M	Alloy
19	10/17	W	Alloy
20	10/22	M	Alcoa

Figure 2: Schedule for analysis of ADTs

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.
Note: There will be no class on 10/19. I will be out of town.

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, 367-375. April 1985.

2. *Markov Analysis of Software Specifications*, J. A. Whittaker and J.H. Poore, ACM Transactions. on Software Engineering Methodology (TOSEM), 2(1), 93-106, January, 1993.

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

4. *Incremental program testing using program dependence graphs*, S. Bates and S. Horwitz, In Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages (POPL), January 1993.

5. *Hierarchical GUI Test Case Generation Using Automated Planning*, A. M. Memon and M. E. Pollack and M. L. Soffa, IEEE Transactions on Software Engineering, 27(2), 144-155, February 2001.

Lecture	Date	Day	Subject
21	10/24	W	CFGs and PDGs [HR92]
22	10/26	F	Data Flow Analysis
23	10/29	M	Data Flow Analysis
24	10/31	W	Testing Metrics
25	11/2	F	Testing Metrics
26	11/5	M	Markov chains
27	11/7	W	Markov Chains
28	11/9	F	Usage based testing
29	11/12	M	Testing GUIs [MPS01]
30	11/14	W	Testing using PDGs [BH93]

Figure 3: Schedule for testing

Tools: None.
Schedule: See Figure 3. Following abbreviations are used:

Control Flow Graphs abbreviated as CFGs.
Program Dependence Graphs abbreviated as PDGS.
Graphical User Interfaces abbreviated as GUIs.

2.4 Program Understanding Tackling Legacy Code

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, IEEE Transactions on Software Engineering, 27(2), 1-25, February 2001.

3. *A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities*, D. Wagner, J. S. Foster, E. A. Brewer and A. Aiken, in Proceedings of the Network and Distributed System Security Symposium (NDSS), February, 2000.

Lecture	Date	Day	Subject
31	11/16	F	Call graphs [CFKW95] and [CGK97]
32	11/19	M	Extracting invariants [ECGN01]
33	11/21	W	Extracting invariants [ECGN01]
34	11/26	M	Slicing [Tip95] and [HR92]
35	11/28	W	Specialized analysis [ECCH00]
36	11/30	F	Specialized analysis [ECCH00] and [WFBA00]
37	12/3	M	Specialized analysis [WFBA00]

Figure 4: Schedule for Program Understanding

4. *Checking System Rules Using System Specific, Programmer-Written Compiler Extensions*, D. Engler, B. Chelf, A. Chou and S. Hallem, in Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, October 2000.

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/`.

Schedule: See Figure 4.
Note: No class on 11/23 due to Thanksgiving break.

2.5 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 5.

2.6 Grading in the course

There will be *no exams* in the course. Regular homeworks will be given in class. Homeworks will account for 40 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. Active participation in class and project will account for 10 percent.

Lecture	Date	Day	Subject
38	12/5	W	Closing comments
39	12/7	F	Comments session

Figure 5: Schedule for Closing

References

[BH93] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages (POPL)*, January 1993.

[BJRR98] 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.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 2001, ACM SIGPLAN Notices*, volume 36, pages 203–213, 2001.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software, LNCS 2057*, pages 103–122, May 2001.

[CAB+98] 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.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, 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.

[CFKW95] Y. Chen, G. Fowler, E. Koutsofios, and R. Wallach. Ciao: A graphical navigator for software and document repositories. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM)*, pages 66–75, 1995.

[CGK97] Y. Chen, E. Gansner, and E. Koutsofios. Data model supporting reachability analysis and dead code detection. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, September 1997.

[DHJ+00] Matthew B. Dwyer, John Hatcliff, Roby Joehanes, Shawn Laubach, Corina S. Pasareanu, Willem Visser, and Hongjun Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, May 2000.

[ECCH00] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA*, October 2000.

[ECGN01] M. D. Ernst, J. Cockrell, W. G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb 2001.

[HP98] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw Hill, October 1998.

[HR92] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE)*, pages 392–411, May 1992.

[MPS01] A. M. Memon, M. E. Pollack, and M. L. Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, February 2001.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[WFBA00] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2000.