Design and Analysis of Communication Software

David L. Dill
Stanford University

Patrice Godefroid
Bell Laboratories

Motivation

- Communication software coordinates the information flow between interconnected components in
  - the telephone network,
  - the internet,
  - wireless networks (cell phones, pagers,…),
  - banking networks,
  - distributed databases (flight reservations,…),
  - etc.
- Communication software is everywhere!
- Communication software is hard to develop and test!

Overview

- "Design and Analysis of Communication Software":
  - Part 1 - The Software Development Process (Sep23)
    - A Roadmap
  - Part 2 - Communication Software (Sep 23)
    - An Introduction
  - Part 3 - Model-Checking Software using VeriSoft (Sep 30)
    - A New Approach to Communication Software Analysis
  - Part 4 - Inside VeriSoft (Oct 14 & 21)
    - The Research Behind The Tool
  - Part 5 - Project: Development and Testing of Simple Internet Phones
    - Application of the Previous Concepts (Oct 28)

Why Me?

- PhD in November 1994 (University of Liege, Belgium)
- Joined Bell Labs in December 1994.
- Bell Labs is part of Lucent Technologies.
- Research on program analysis, testing and verification.
- Developed several tools (parts of SPIN & VFSMvalid, VeriSoft).
- Applied these tools to industrial protocols and software.
- Exposed to technology-transfer and business issues.
- Why you?

Logistics

- This is an experimental course (="notes ready at last minute" ;-).
- Please do not hesitate to ask questions!
- Lectures open to the public, credits for the project (talk to Dave).
- Lectures given on Thursday morning 9am-noon, Gates B12.
  - Schedule, slides and other relevant info will be posted on the course web-page: http://sprout.stanford.edu/comm.html
- Project due at end of term; need account on Sun machines.
- Emergency: god@bell-labs.com

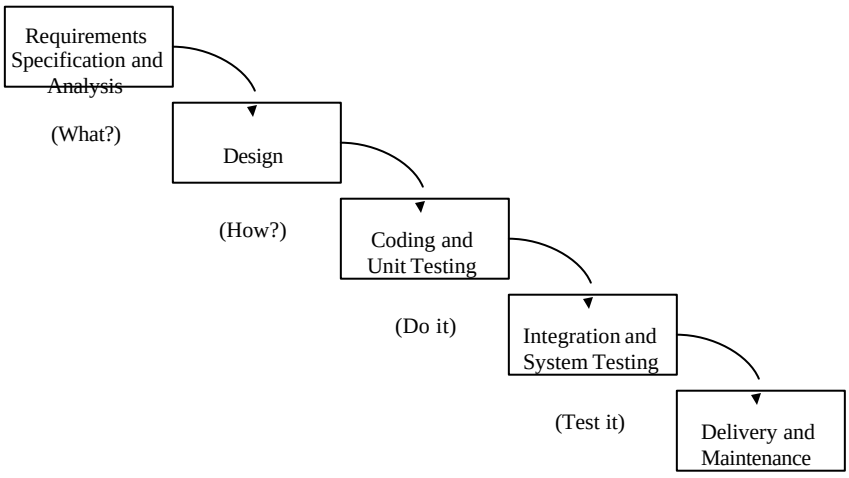
Part 1: The Software Development Process

A Roadmap

Overview

- Main steps of the software development process.
- Main tools used for each of these steps in industry today.
- More detailed discussion on testing.

The Waterfall Model



Note: in real-life, steps overlap and have feedback loops!

Steps

- Requirements Specification and Analysis:
  - Determine customer-visible features, feasibility study, development costs and price of product.
  - Determine “what to do”.
  - Done by “system engineers”.
- Design:
  - Determine high-level and detailed design of product that will meet the requirements.
  - Determine “how to do it”.
  - Done by “architects”.

Steps (continued)

- Coding and Unit Testing:
  - Produces the actual code that will be delivered to the customer, and test individual modules in isolation.
  - Done by “developers” (“programmers”).
- Integration and System Testing:
  - Test the integration of individual modules and the whole system.
  - Done by “testers”. Note: testing implies running the code.
- Delivery and Maintenance:
  - Deliver the product to the customer and provide documentation, training, field support, and bug fixes.
- “Product Manager”: manages the end-to-end process.

Development Tools

- What are the most common tools used at each step of the development process today in the software industry?
- Warnings:
  - The list that follows is not exhaustive and is also based on the experience of the speaker.
  - Only general-purpose tools are considered, not application-specific tools (for GUI, web, databases,…).
  - Names of commercial products are used only as examples, for illustration purposes only.

Tools for Requirements Specification and Analysis

- MSWord and PowerPoint…
  - Requirements are often imprecise, ambiguous and incomplete.
  - This can be done partly on purpose...
- “Formal Notations”:
  - Use-cases, state-machines, Message Sequence Charts, tables, decision trees,…
  - Less ambiguous than English text.
  - Enable simple automatic analysis of specification (check for consistency).
  - Can only cover a subset of the requirements.
  - In practice, used in conjunction with English text.

Tools for Design

- MSWord and PowerPoint…
  - with diagrams, tables, state-machines, Message Sequence Charts,…
- Modeling Languages: (for design and high-level coding)
  - UML, SDL, ObjecTime, VFSM,...
  - Less ambiguous than English text.
  - Enable automatic analysis.
  - Can be executed.
  - Automatic code generation of a template of the implementation.

Tools for Coding

- Defect Tracking & Resolution Managers:
  - Track problem reports and the status of their resolution.
  - Record “history” of the system.
  - Also used by testers and during maintenance.
- Version Control Systems:
  - Controls and coordinates the various versions of the software (SCCS,…).
- Code Browsers and Editors:
  - help navigate through the code,
  - and through the history of the code.
  - Help compares different versions of the code (diff).

Tools for Coding (Continued)

- Compilers:
  - Translate (higher-level) source language to (lower-level) target language.
  - Report syntax errors.
- Linkers:
  - Combine mutually referencing object-code fragments; report errors at module interface.
- Code Reviewers (= Static Analyzers):
  - examine source code, detect programming errors, provide suggestions on code structure and style (advanced type checkers, Lint,…).
  - Automated tools for detecting semantic errors through (local) symbolic execution (Prefix,…).
  - **Colleagues**!

Tools for Testing

- Debuggers:
  - Requires code instrumentation (usually during compilation).
  - Control and examine code execution.
- Memory Analyzers:
  - Detect memory leaks and overflows:
    - memory leak = memory allocated, no more reachable but not freed.
    - memory overflow: access to unauthorized memory address (unallocated/uninitialized memory, array out-of-bounds,…).
  - Parse and instrument source or binary code to check properties at run-time.

Tools for Testing (Continued)

Lucent Technologies
Bell Labs Innovations

- Performance Analyzers (Profilers) and Code Coverage Tools:
  - Count number of occurrences of executions of program statements or procedures.
  - Report time spent in each part of program during execution.
  - Parse and instrument source or binary code to record run-time information.
- Languages and platforms for test automation:
  - Example: expect…
- Capture/Replay Tools:
  - Record/replay actions performed during manual testing at standard interface (e.g., GUI/web testing).

Tools for Testing (Continued)

Lucent Technologies
Bell Labs Innovations

- Load Generators:
  - Simulate environment (e.g., traffic) through standard interface.
- Test Case Generation from Specification:
  - Generate sets of tests from higher-level specification of I/O behavior.
  - Easier test management, better coverage.
- Test Management Tools:
  - Process: help record test plans, track and report the status of testing project.
  - Code: store and execute test code, compare and store results.
  - Used by the testing organization only.

More on Testing

- Why test? "To find errors."
  - "The process of executing a program with the extent of finding errors." [Myers,1979]
- What is an "error"? "Any problem visible to the customer."
  - Programming errors, conflicts with requirements, unexpected behaviors, features too hard to use, etc.
- When to stop testing?
  - In theory, when full coverage is reached!
  - Coverage can be defined versus requirement, formal I/O spec, code or state-space.
  - In practice, test until shipment date!

Tools for Testing: Summary

- Three main types of tools for testing:
  - 1. Code Inspection:
    - analyses (parses) the code to find programming errors.
  - 2. Code Instrumentation:
    - analyses (parses) source code or binary code and inserts code (such as assertions) to check properties at run-time.
  - 3. Code Execution:
    - help generate, execute and evaluate tests performed by running the code in conjunction with a representation of its environment.
- Type 2 and 3 are complementary.
- In practice, Type 1 also complementary with Types 2 and 3.

Testing World

- Level 1: Manual Testing
  - Most testing organizations; some tests cannot be automated.
- Level 2: Automated Testing
  - Automated test execution and evaluation.
  - Advantage: automated regression testing.
- Level 3: Automatic (Static) Test Generation
  - Automated test generation from higher-level spec.
  - Advantage: easier test management, better coverage.
- Level 4: Automatic Dynamic Test Generation
  - VeriSoft! See later…

Some References

- Software Engineering: A Practitioner's Approach, R.S. Pressman, McGraw-Hill.
- Fundamentals of Software Engineering, C. Ghezzi, M. Jazayeri and D. Mandrioli, Prentice-Hall.
- Software Testing in the Real World, E. Kit, Addison-Wesley.
- Surviving the Challenges of Software Testing, W.E. Perry and R.W. Rice, Dorset House.

Summary of Part 1

- Main steps in the software development process.
  - Requirements, Design, Coding, Testing, Maintenance.
- Tools used in each of these steps.
- Taxonomy of tools used for testing.
  - Code Inspection, Instrumentation and Execution.

Part 2: Communication Software

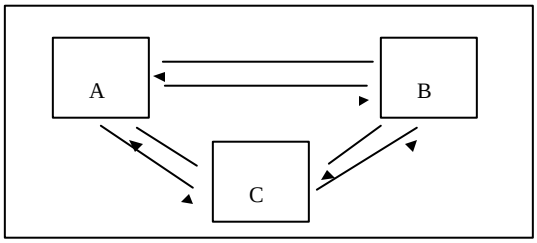
An Introduction

Overview

- Specificity of communication software.
- Specific tools.
- A new approach: model checking software.

What is Communication Software?

- Coordinates the information flow between interconnected components.



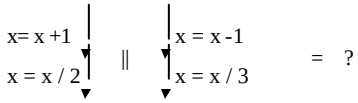
- Each component can be viewed as a reactive system (continually interacts with its environment).
- Examples: telephone, internet, wireless, banking,...

Originality

- Communication software is software!
  - Same overall development process and general-purpose tools (see Part 1).
- Developing communication software is harder!
  - Many possible sequences of interactions between components (coordination problems, race conditions, timing issues,...).
- Testing communication software is harder!
  - Traditional testing provides poor coverage.
- Debugging communication software is harder!
  - Scenarios leading to errors can be hard to reproduce.

Why Harder?

- Implementation looks nondeterministic due to concurrency (scheduling) and real-time (processing speed):
  - "Nondeterministic" means "unpredictable"; it is an abstraction.
  - Same sequence of inputs does not imply same sequence of outputs.
- Fundamentally, parallel composition is not "compositional":
  - Given 2 functions f(x) and g(x), f(g(x)) is easy to understand.
  - Example: if f(x)=(x+1)/2 and g(x)=(x-1)/3, f(g(x))=(((x-1)/3)+1)/2
  - Given 2 functions f(x) and g(x), f(x)||g(x) can be very different from f or g!
  - Example:



Tools for Dealing with Concurrency

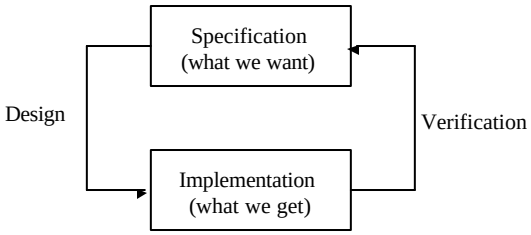
- Debuggers for concurrent/distributed systems:
  - Control and track the execution of more than one process/thread.
- Tools for detecting run-time coordination problems:
  - Detect race conditions (simultaneous writes in same address) and coordination problems (deadlocks) at run-time.
  - Instrument the execution of processes/threads while minimizing the impact on timing.
  - Record scheduling information ("trace") for faithfully replaying multi-process scenarios leading to errors.
  - Generate a consistent representation (snapshot) of the state of a distributed system.
  - Example: Eraser, Assure (for Java)

Tools for Dealing with Real-Time

- Schedulability Analyzers:
  - Analyze a set of real-time scheduling constraints (coming from architecture and properties to satisfy) and generate a schedule if there exists one.
- Worst-Case Execution-Time Analyzers:
  - Determine WCET of fragments of code.
- Performance modeling tools:
  - Analyze performance of an architectural model (queuing theory, stochastic processes,...)
- Etc.

Another Approach: Formal Verification

- What is Verification? 4 elements define a verification framework:



Verification: to check if all possible behaviors of the implementation are compatible with the specification

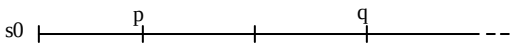
- While testing can only find errors, verification can also prove their absence (=exhaustive testing).
- Examples of Approaches: Theorem Proving and Model Checking.

Theorem Proving

- Goal: automate mathematical (logical) reasoning.
- Verification using theorem proving:
  - Implementation represented by a logic formula I.
  - Specification represented by a logic formula S.
  - Does “I implies S” hold?
  - Proof is carried out at syntactic level.
- This framework is very general.
  - Many programs and properties can be checked this way.
- However, most proofs are not fully automatic.
- A theorem prover is rather a proof assistant and a proof checker.

Model Checking

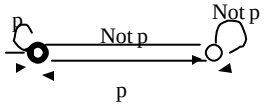
- Model Checking is more restricted in scope but is fully automatic.
- Verification using model checking:
  - Implementation represented by a finite state machine M (called state space)
  - Specification represented by a temporal-logic formula f.
    - Example: Linear-time Temporal Logic (LTL)
      - Specify properties of infinite sequences s0, s1, s2,… of states
      - Temporal operators include: G (always), F (eventually) and X (next).
      - Example: G(p -> Fq)



  - Does "M satisfies f" hold? (Hence the term "model checking"...)
    - For LTL, do all infinite computations of M satisfy f?
  - Proof is carried out at semantic level, via state-space exploration.

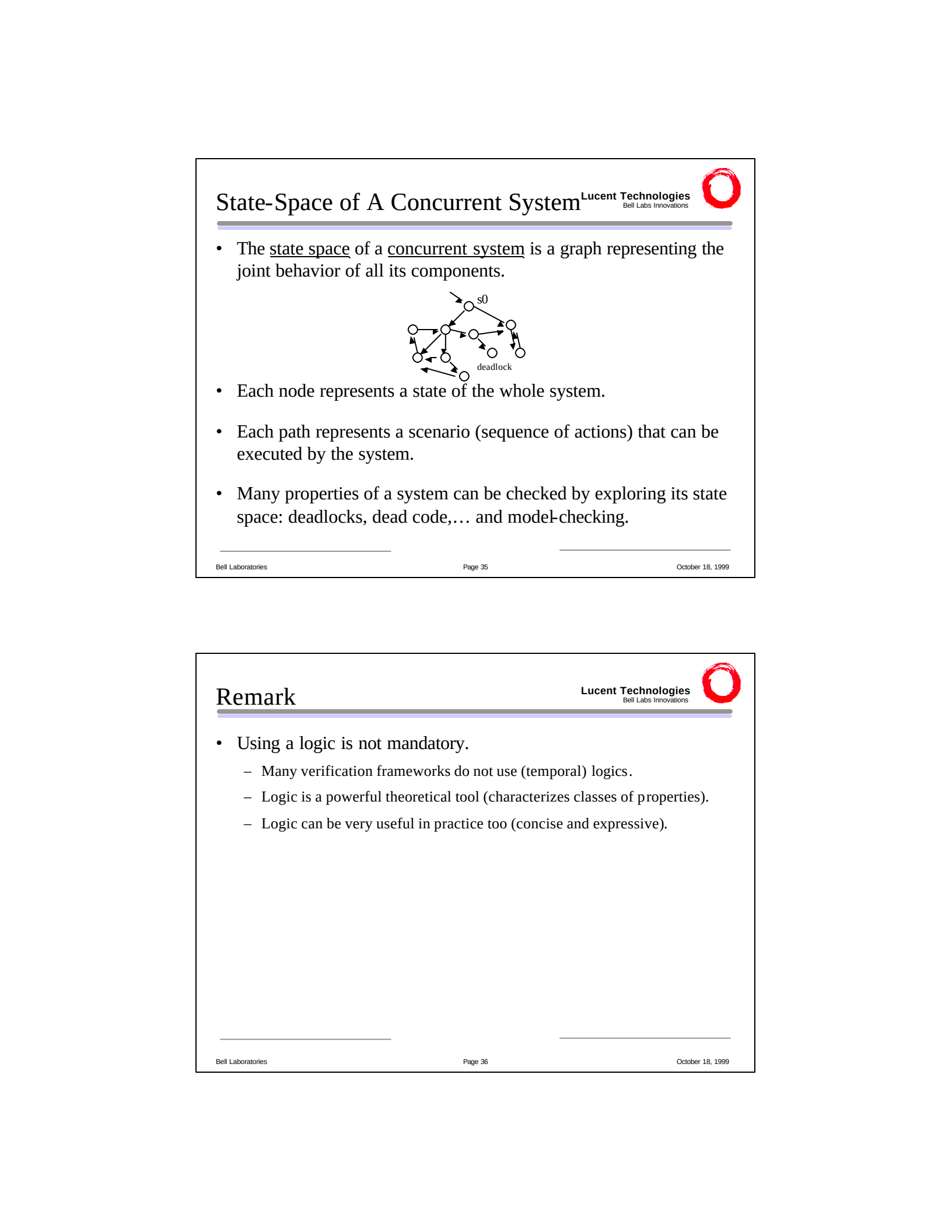
Model Checking Procedure for LTL

- Property: Every LTL formula f can be translated into a finite automaton on infinite words A(f) (Buchi automaton) such that A(f) accepts exactly the infinite sequences satisfying f.
  - Example: automaton accepting GFp



- Model Checking Procedure: ("automata-theoretic approach")
  - 1. Build A(Not f) (the size of A(Not f) is at worst exponential in |Not f|).
  - 2. Compute the product automaton of the state space and A(Not f).
  - 3. Check that the product automaton is empty (linear-time complexity).

State-Space of A Concurrent System



- The state space of a concurrent system is a graph representing the joint behavior of all its components.

- Each node represents a state of the whole system.
- Each path represents a scenario (sequence of actions) that can be executed by the system.
- Many properties of a system can be checked by exploring its state space: deadlocks, dead code,… and model-checking.

Remark

- Using a logic is not mandatory.
  - Many verification frameworks do not use (temporal) logics.
  - Logic is a powerful theoretical tool (characterizes classes of properties).
  - Logic can be very useful in practice too (concise and expressive).

The State Explosion Problem

- Main limitation: “state explosion” problem!

  - State-space exploration is fundamentally hard (NP, PSPACE or worse).
  - Divide-and-conquer approaches:
    - abstraction: hide/approximate details.
    - compositionality: check first local properties of individual components, then combine these to prove correctness of the whole system.
  - Algorithmic approaches:
    - “symbolic verification”: represent state space differently (BDDs,…).
    - state-space pruning techniques: avoid exploring parts of the state space (partial-order methods, symmetry methods,…).
    - Etc.

Summary

- Systematic State Space Exploration is simple:
  - easy to understand,
  - easy to implement,
  - easy to use: automatic!
- Main limitation: “state explosion” problem.
- Used in many tools: CAESAR, COSPAN, MURPHI, SMV, SPIN, etc.
  - Differ by specification language, implementation language, comparison criterio, and/or verification algorithms,
  - but all based on systematic state-space exploration.

Formal Verification vs. Testing

- Experiments with these tools show that model checking can be very effective!
  - They can detect subtle design errors.
- In practice, formal verification is actually testing because of approximations:
  - when modeling the system,
  - when modeling the environment,
  - when specifying properties,
  - when performing the verification.
- Therefore “bug hunting” is really the name of the game!

Applications: Hardware

- Hardware verification is a booming application of model checking and related techniques.
  - The finite-state assumption is not unrealistic for hardware.
  - The cost of errors can be enormous (e.g., Pentium bug).
  - The complexity of designs is increasing very rapidly (system on a chip).
- However, model-checking still does not scale very well.
  - Many designs and implementations are too big and complex.
  - Hardware description languages (Verilog, VHDL,…) are very expressive.
  - Using model checking properly requires experienced staff.
- Quid for Software?

Applications: Software Models

- Analysis of software models: (e.g., SPIN)
  - Analysis of communication protocols, distributed algorithms.
  - Models specified in extended FSM notation.
  - Restricted to design.
- Analysis of software models that can be compiled: (e.g., SDL, VFSM)
  - Same as above except that FSM can be compiled to generate the core of the implementation.
  - More popular with software developers since reuse of “model” is possible.
  - Analysis still restricted to “FSM part” of the implementation.

Applications: Software

- Quid for software?
  - General-purpose programming languages (e.g., C, C++, Java),
  - Real size (e.g., hundred thousand lines of code).
- Two main approaches for software model checking:

Model Checking Software

- Static analysis for automatic model extraction: (e.g., ?)
  - Language dependent +often need additional restrictions (heavy machinery).
  - Abstraction is not a panacea: it always introduces unrealistic behaviors.
  - Need to map scenarios leading to errors back to the code.
  - Technology not ready yet, active area of research.
- Systematic state-space exploration for arbitrary code: **VeriSoft**
  - Controls the execution of concurrent processes by intercepting systems calls related to communication.
  - Automatically drive the entire system through many scenarios.
  - Provide a complete state-space coverage up to some depth only.
  - VeriSoft is not a panacea either, but it is available today!

VeriSoft

- What is it?
- How does it work?
- Existing Industrial Applications?
- Comparison with Testing?
- Challenges?
- How to use it?

… See you next week!

Summary of Part 2

- Developing, testing and debugging communication software is hard.
- Alternative approach: formal verification and model checking (=systematic state-space exploration).
- Model checking software.