Lecture 27 — Verification & Validation; Software Maintenance

Patrick Lam & Jeff Zarnett
p.lam@ece.uwaterloo.ca & jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 29, 2014

Part I

Verification and Validation

Verification vs validation

Two similar terms. Not the same.

- Verification: “Is the project being built correctly?”
- Validation: “Will the project meet users’ needs?”

Need both!

Projects can fail because:

- meet users’ needs, but don’t work right; or
- bug-free but solve the wrong problem.

A successful project must pass both verification and validation.

More on Verification

Verification:

- assume a set of requirements; and
- establish that the product satisfies the requirements.

The requirements might be wrong. Not our problem!

How to verify: two options—

- testing (as seen previously);
- static analysis
  (computers exhaustively check code/design).

“Building the thing right.”

More on Validation

Validation:

make sure that the requirements are the right ones.

(How does XP incorporate validation?)

Go beyond checking that code meets specifications;
work with customer to ensure specifications are correct.

“Building the right thing.”

One way to validate code: *beta testing*.

- customers try the software and see if it’s good.

Which first?

Verification, then validation.

Validating buggy software is frustrating.
Too much verification can be wasted work.

Independent V & V

Sometimes V&V processes are carried out by a different team.

This is known as Independent Verification and Validation.

Common in cases where there is extreme expense or risk to human life or health.

NASA established IV&V in 1993.

Regulatory Environment

Validation might be required given a regulatory environment.

The US FDA requires software patches to be validated for networked medical devices.

The device manufacturer bears responsibility for the safe and effective performance of the device.

FDA will review when a change or modification could significantly affect the safety or effectiveness of the device.

Part II

Formal Methods

About Formal Methods

Formal methods: techniques for verification:
    make sure that code/designs conform to a specification.

Some formal methods techniques:

- static analysis;
- model checking.

Key Idea

To use formal methods, you need:

- a model of the artifact in question; and,
- a property that you would like to verify.

Often, the model is an abstract graph (ECE103!) representing system behaviours.

The property is usually a temporal logic formula.

Verification exhaustively searches model for violations.

Result of Verification

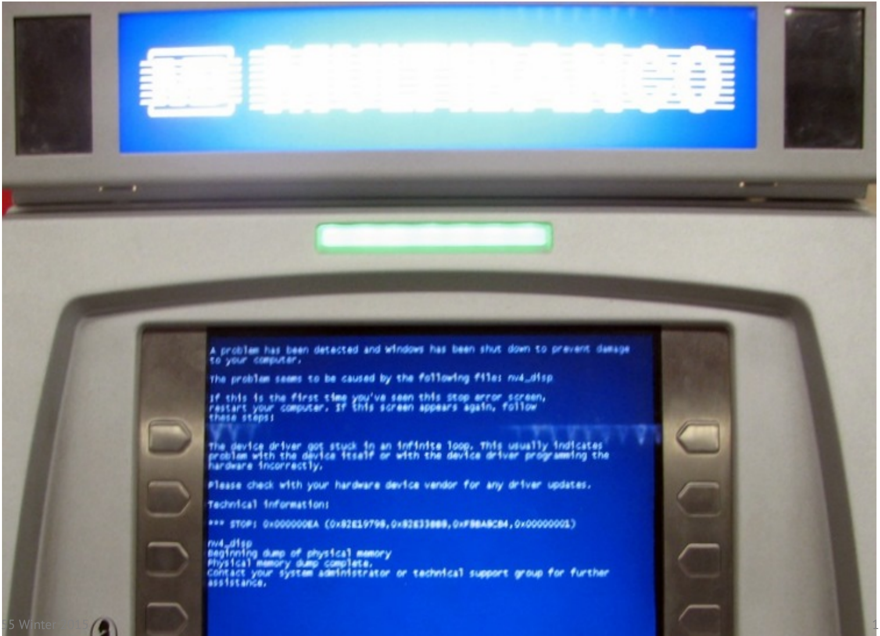
After the exhaustive search, either:

- the property definitely holds (on the model); or
- you get a counterexample.

With cleverness, we can search huge state spaces (10^{100} states).

Main insight: leveraging symmetries.

Case Study: Microsoft Static Driver Verifier



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

The problem seems to be caused by the following file: nv4_disp

If this is the first time you've seen this stop error screen,
restart your computer. If this screen appears again, follow
these steps:

The device driver got stuck in an infinite loop. This usually indicates
problem with the device itself or with the device driver programming the
hardware incorrectly.

Please check with your hardware device vendor for any driver updates.

Technical information:

*** STOP: 0x000000EA (0x82E19798,0x82E33888,0xF8BA9CB4,0x00000001)

nv4_disp
Beginning dump of physical memory
Physical memory dump complete.
Contact your system administrator or technical support group for further
assistance.
```

Why does Windows crash?

Usually, not the Windows kernel—fairly bombproof now.
Instead: Windows drivers; run at same protection level as the kernel.

Scientists at Microsoft Research integrated existing and new techniques to verify drivers.

Windows Driver Kit includes the Static Driver Verifier;
any “Certified for Windows” product must pass the SDV.

Formal Methods in Action

Recall: formal methods tools *exhaustively* explore all possible states of the system and driver.

SDV knows about all of the ways that the operating system can call the driver, and (symbolically) tries all possible combinations of calls.

Formal Methods: Discussion

In general, formal methods tools take a long time to run. Experts must give hints to tools.

Problems:

- need to wait too long for a result;
- can’t verify something that is correct;
- false positives: warnings about problems that can never happen.

False positives occur when the model is too coarse and includes cases that can never happen in practice.

Formal methods particularly useful when the problem domain is too hard to reason about manually,

e.g. concurrency.

Part III

Software Maintenance

On Software Maintenance

Who ever heard of a Fourth Year Maintenance Project?

Yet, in the real world, maintenance accounts for much engineer effort.

On Software Maintenance

Software maintenance modifies existing software to fix defects, improve performance, or make the software work in new environments (porting).

Software Maintenance is Hard

Why?

- must understand the existing code, which can be difficult. (whether code is yours or someone else's!)
- it's unglamorous, especially if you're fixing bugs.
- it's constrained; you better not break compatibility.

The Problem with Patches

Patching can lead to gnarly code with no design.
Question: What's the alternative?

Temptation: start over from scratch.

- Sometimes, existing software looks hopeless.
- It's more fun to redesign rather than maintain.

Another Story from Ancient History (or: Motivation)

It’s harder to read code than to write it.

Joel Spolsky

Another Story from Ancient History (or: Motivation)

Another Story from Ancient History (or: Motivation)

Netscape Navigator was a web browser (forerunner of Firefox).

Back in the year 2000, Netscape decided to rewrite their browser totally from scratch.

It took them three years.

Their market share collapsed and Microsoft Internet Explorer dominated the internet – a disaster for standards compliance.

Maintenance: Rewrite?

Programmers often say: this code is a mess!

Functions that are two pages long; 14-if statements; a thousand function calls...

Plus it's old, and newer is better, right?

Maintenance: Rewrite?

Code does not degrade as it gets older.

Unlike buildings where concrete crumbles and steel rusts.

Old code can be better:

It's been tested.

Bugs have been fixed in it.

Bugs don't appear magically in code.

Maintenance: Rewrite?

That function that has 14 if statements and weird stuff?

Those oddities are bug fixes. Some possible cases handled:

- $<$ 1GB RAM is available.
- The default temp directory is not writeable.
- The user is (for unknown reasons) still using Windows 98.

Each of these probably took a lot of testing to identify and fix.

Maintenance: Don’t Rewrite

Maybe the code has serious architectural problems.

Solution: refactoring!

Is it slow?

Solution: Take the class Programming for Performance.

Maintenance: Don’t Rewrite

When you start from scratch, you are throwing away a lot of collected knowledge and bug fixes.

“Second system effect”: you may do worse by starting over.

Software Maintenance: Beyond Bug-Fixing

Per T. M. Pigosky[1], approximately 80% of software maintenance activities are unrelated to defect fixes.

Types of maintenance for already-shipped code:

- Corrective Maintenance: correct known defects;
- Adaptive Maintenance: keep a software product usable in a changing environment;
- Perfective Maintenance: improve performance or maintainability; and
- Preventive Maintenance: correct latent faults in the product before they manifest themselves.

[1] T.M. Pigosky, *Practical Software Maintenance*, John Wiley & Sons, 1997.

Corrective Maintenance Example

You come across a piece of code that reads
`getAppliance().activate(item);`

This results in a `NullPointerException` if `item` is `null`.

Corrected by adding a null-check around that statement:

```
if (item != null) {
     getAppliance().activate(item);
}
```

Adaptive Maintenance Example

Conform to new security and data storage guidelines when Android 5.0 is released.

Sometimes it’s optional (OS upgrades rarely “forced”).

Sometimes no choice: government will no longer accept tax declarations via http starting from 1 January.

Perfective Maintenance Example

Refactoring, but not limited to that.

Query some records from the database; sort them in-memory.

Improve by having the database sort the results.

Preventive Maintenance Example

Rare, but it does happen.

“Y2K” Bug - 2 digit dates + year 2000 = errors.

Corrected in advance by using 4 digit dates.

(But then we’ll face the Y10K problem).

Bugs vs. New Features

The lines between the different categories are unclear.

Users love to report “bugs” that are really feature requests.

Sometimes, but not always, it’s obviously a bug (e.g., a `NullPointerException` is thrown).

Bugs vs. New Features

Users don’t care about the distinction between bug & feature.

They want to do something & the software doesn’t support it.

The cause is unimportant to users.

Bugs vs. New Features

Sometimes there are financial implications.

- Users can be charged for new features.
- They are unlikely to pay for bug fixes.

Bug fixes also have to be patched into released software.

Managing Maintenance

Software projects are huge.

Bugs are everywhere, even in shipped software.

10 000s of defects are common.
(Average bug lifetime in Linux: 1.38 years.)

Key to avoiding analysis paralysis: triage.

Some bugs are more important than others;

- security fixes—pushed right away;
- minor defects can wait (perhaps forever).

Managing Maintenance

When there are multiple versions, decide which branches.

At some point, moving a fix into an old version is not worth it.

Release Manager

Some projects have a release manager.

A release manager must be knowledgeable about software engineering in general

Takes responsibility for:

- Assembling all the various updates,
- deciding when to make a new release,
- troubleshoot problems caused by an update

Patch Discipline

Changes = potential problems.
Negative progress is always possible.

Before pushing a change,
check that it makes things overall better.

Testing is particularly critical. Also:

- reviews;
- regression tests;
- other verification techniques.

Do less harm than good.

Maintenance Agreements

When large organizations buy (or license) some software, they often want to have a signed maintenance agreement.

The agreement often covers:

- What constitutes a defect in the software?
- What types of defect, if any, are not covered?
- How to categorize a defect (priority 1, 2, etc)?
- Who is responsible for categorization?
- What is the flexibility of categories/ ability to escalate?
- What are the response times and resolution times promised?

Maintenance Agreements

- What are the support hours, and what are the overtime charges, if any?
- Will the supplier be expected to travel to the customer to fix defects?
- Are new releases included or just patches?
- Will old versions cease to be supported at some point?
- Can the customer delay installation of a patch?
- How regularly are upgrades to be provided?
- How are charges and payment settled?

Retirement

One day, successful software is retired.

Support and development are discontinued.

This is usually a business decision.

Sometimes one final maintenance project: export data.