Lecture 34 — April 5, 2013

Patrick Lam *version 1*

Verification and Validation

It's important to not confuse the two related topics of verification and validation. In *verification*, we assume a set of requirements and establish that the product satisfies the requirements. The requirements might be wrong, but we can put up a "Someone Else's Problem" field and say that it's not our problem when doing verification. *Validation* is making sure that the requirements are the right ones.

Verification. We've already seen one form of verification: *testing*. The other option is *static analysis*, which amounts to having computers pore over the code or design. ("Building the thing right.")

Validation. Validation ensures that a project fully satisfies the needs of its customers. (How does XP incorporate validation?) Validation should therefore go beyond checking that code meets specifications and work with the customer to ensure that the specifications are correct. ("Building the right thing.")

One way to validate code is through *beta testing*: customers try out early versions of the software and determine whether or not it is doing the right thing.

Validation usually follows verification. It's not productive to validate buggy software, but taking too long in verification can result in validating high-quality code that satisfies the wrong specification.

Recap. The terms are similar, but:

- Verification answers the question, "Is the project being built correctly?"
- Validation answer the question, "Will the project meet the needs of its users?"

You need to verify *and* validate; projects can fail because they're properly-built but don't meet needs; or they can sort-of meet users' needs except that they don't work right.

A successful project must pass both verification and validation.

Formal Methods

Formal methods includes a variety of techniques for verifying that code or designs conforms to a specification. Static analysis is a formal methods technique, usually at code, or implementation, level. It's possible to formally verify system designs as well as implementations.

Key Idea. To use formal methods, you need a model of the artifact in question, along with a property that you would like to verify. The model is often some sort of abstract graph representing system behaviours, while the property is usually some sort of (temporal logic) formula. Verification exhaustively searches the model for violations of the property; it either tells you that the property holds, or it shows you a counterexample. By being clever, it's possible to verify huge state spaces (e.g. 10^{100} states). In particular, leveraging symmetries allows the verification to drastically reduce the required search space.

Case Study: Microsoft's Static Driver Verifier. Windows historically has a bad reputation for producing blue screens of death.

It turns out that most Windows crashes these days are caused not by the Windows kernel itself, which is fairly bombproof, but instead by Windows drivers, which run at the same protection level as the kernel.

Scientists at Microsoft Research have integrated existing techniques and new techniques to verify drivers. The Windows Driver Kit includes the Static Driver Verifier[1], and any "Certified for Windows" product needs to pass the SDV.

Formal methods tools *exhaustively* explore all possible states of the system and driver. In particular, the SDV knows about all of the ways that the operating system can call the driver, and (symbolically) tries out all of the combinations of calls that can occur.

Discussion. In general, formal methods tools are still for experts (i.e. not you, at least today). Experts need to give hints to the tools so that they run in some reasonable amount of time.

Besides having to wait too long to get a result, and not being able to get the verification to go through, the main shortcoming with formal verification is that you can get (way too many) false positives: the verification warns you of a problem that can never happen. This occurs when the model is too coarse and includes cases that can never happen in practice.

Formal methods are particularly useful when the problem domain is too hard to reason about manually. Concurrency is a good example of such a domain.

[1]For more information, see `http://msdn.microsoft.com/en-us/windows/hardware/gg487506`.