Varieties of methods of verification [H&R, Chapter 4, 256-258

Introduce Hoare logic to illustrate a style of program verification:

- aimed at verification of *sequential* processes running on a single processor
- may manipulate non-trivial data, possibly with variables of type integer, list or tree
- infinite state space

Contrasts with other techniques (such as *computation tree logic* discussed by H&R in Chapter 3) that are aimed at concurrent systems, where the emphasis is on control and communication, and the models are finite-state based.

Characteristics of the application in this context are:

- **Proof-based**
  Not model-checking as program variables can have infinitely many values. Aim to construct proofs instead.

- **Semi-automatic**
  Some mechanical steps for which automated support can be provided, but human intelligence is essential.

- **Property-oriented**
  Verify properties of a program, rather than a full specification of its behaviour.

- **Application domain**
  Targeting sequential transformational programs. In contrast to reactive systems that are not intended to terminate and react continuously to their environment.

- **Pre-post development**
  Techniques to be described are to be used during the coding process for small fragments of programs that perform identifiable (hence specifiable) tasks. Aimed at eliminating functional bugs (not concerned with non-functional requirements).

Motivations for verification proposed by H&R:

- **Documentation**
  The specification of a program serves as an important component of its documentation.

- **Time-to-market**
  Verifying programs with respect to formal specifications can significantly reduce the development time. Cf. debugging big systems during the testing phase, when fixing one bug can introduce others.

- **Reuse**
  Properly specified and verified code is easier to reuse.

- **Certification audits**
  Safety-critical / commercial-critical systems demand that software be specified and verified with as much rigour and formality as possible.

Microsoft's A# as an example of an emergent technology that "combines program verification, testing and model-testing techniques in an integrated in-house development environment".

Capability Maturity Model for Integration (CMMI) models for software development developed at the Software Engineering Institute, Carnegie Mellon.

A framework for software verification [H&R, 4.2]

Real-world task such as "the development and maintenance of a DB of electricity accounts with all the possible

applications of that - automated billing, customer service etc."

"desirable to condense all the requirements into formal specifications ... usually symbolic encodings of real-world constraints into some sort of logic"

Strategy:

- convert the informal description of the requirement R into an 'equivalent' formula Φ_R of some symbolic logic
- Write a program P that is meant to realise Φ_R in the appropriate programming environment
- *Prove* that executing your program P satisfies the formula Φ_R ...

H&R observe:

> ... 'going back and forth' between the realms of informal and formal specifications is necessary since it is impossible to 'verify' whether an *informal* requirement R is equivalent to a *formal* description Φ_R
>
> ...to make matters worse, the requirements R are often inconsistent; customers usually have a fairly vague conception of what exactly a program should do for them.

A core programming language

Three syntactic domains: integer expressions (E), boolean expressions (B) and commands (C). The intuitive meanings of the constructs in the programming language are as follows:

1. **Assignment**
   The atomic command `x = E` is the usual assignment statement.

2. **Composition**
   The compound command C_1`;` C_2 is the sequential composition of commands C_1 and C_2. If the executon of C_1 does not terminate, then C_1`;` C_2 does not terminate.

3. **`if`-construct**
   The control structure `if B{`C_1`} else {`C_2`}` is interpreted in such a way that it first evaluates the boolean expression `B` in the current state of the store, then executes the command C_1 or C_2 according to whether `B` is true or false.

4. **`while`-construct**
   The control structure `while B {C}` is interpreted in such a way that it first evaluates the boolean expression `B` in the current state of the store, terminating if `B` is false, and repeatedly executing the command `C` while `B` is true.

A simple example program

The following program, called `Fac1`:

```
y = 1;
z = 0;
while (z != x) {
        z = z+1;
        y = y*z;
}
```

is intended to compute the factorial x! of a natural number x.

A challenge: Can we prove such a program to be correct? This motivates Hoare logic.
