Program verification

Why should we specify and verify code?

Documentation: The specification of a program is an important component in its documentation and the process of documenting a program may raise or resolve important issues. The logical structure of the formal specification, written as a formula in a suitable logic, typically serves as a guiding principle in trying to write an implementation in which it holds.

Time-to-market: Debugging big systems during the testing phase is costly and time-consuming and local 'fixes' often introduce new bugs at other places. Experience has shown that verifying programs with respect to formal specifications can significantly cut down the duration of software development and maintenance by eliminating most errors in the planning phase and helping in the clarification of the roles and structural aspects of system components.

Refactoring: Properly specified and verified software is easier to reuse, since we have a clear specification of what it is meant to do.

Certification audits: Safety-critical computer systems – such as the control of cooling systems in nuclear power stations, or cockpits of modern aircrafts – demand that their software be specified and verified with as much rigour and formality as possible. Other programs may be commercially critical, such as accountancy software used by banks, and they should be delivered with a warranty: a guarantee for correct performance within proper use. The proof that a program meets its specifications is indeed such a warranty.

The degree to which the software industry accepts the benefits of proper verification of code depends on the perceived extra cost of producing it and the perceived benefits of having it. As verification technology improves, the costs are declining; and as the complexity of software and the extent to which society depends on it increase, the benefits are becoming more important. Thus, we can expect that the importance of verification to industry will continue to increase over the next decades. Microsoft's emergent technology A# combines program verification, testing, and model-checking techniques in an integrated in-house development environment.

Currently, many companies struggle with a legacy of ancient code without proper documentation which has to be adapted to new hardware and network environments, as well as ever-changing requirements. Often, the original programmers who might still remember what certain pieces of code are for have moved, or died. Software systems now often have a longer life-expectancy than humans, which necessitates a durable, transparent and portable design and implementation process; the year-2000 problem was just one such example.

A framework for software verification

Suppose you are working for a software company and your task is to write programs which are meant to solve sophisticated problems, or computations. Typically, such a project involves an outside customer – a utility company, for example – who has written up an informal description, in plain English, of the real-world task that is at hand. In this case, it could be the development and maintenance of a database of electricity accounts with all the possible applications of that – automated billing, customer service etc. Since the informality of such descriptions may cause

ambiguities which eventually could result in serious and expensive design flaws, it is desirable to condense all the requirements of such a project into formal specifications. These formal specifications are usually symbolic encodings of real-world constraints into some sort of logic. Thus, a framework for producing the software could be:

Convert the informal description R of requirements for an application domain into an 'equivalent' formula φR of some symbolic logic;

Write a program P which is meant to realise φR in the programming environment supplied by your company, or wanted by the particular customer;

Prove that the program P satisfies the formula φR. This scheme is quite crude – for example, constraints may be actual design decisions for interfaces and data types, or the specification may 'evolve

A core programming language

The programming language which we set out to study here is the typical core language of most imperative programming languages. Modulo trivial syntactic variations, it is a subset of Pascal, C, C++ and Java. Our language consists of assignments to integer- and boolean-valued variables, ifstatements, while-statements and sequential compositions. Everything that can be computed by large languages like C and Java can also be computed by our language, though perhaps not as conveniently, because it does not have any objects, procedures, threads or recursive data structures. While this makes it seem unrealistic compared with fully blown commercial languages, it allows us to focus our discussion on the process of formal program verification. The features missing from our language could be implemented on top of it; that is the justification for saying that they do not add to the power of the language, but only to the convenience of using it. Verifying programs using those features would require non-trivial extensions of the proof calculus we present here. In particular, dynamic scoping of variables presents hard problems for program-verification methods, but this is beyond the scope of this book. Our core language has three syntactic domains: integer expressions, boolean expressions and commands – the latter we consider to be our programs. Integer expressions are built in the familiar way from variables x, y, z,... , numerals 0, 1, 2,..., −1, −2,... and basic operations like addition (+) and multiplication (∗). For example,

$$5$$
$$x$$
$$4 + (x - 3)$$
$$x + (x * (y - (5 + z)))$$

are all valid integer expressions. Our grammar for generating integer expressions is

$$E ::= n \mid x \mid (-E) \mid (E + E) \mid (E - E) \mid (E * E)$$

where n is any numeral in {..., −2, −1, 0, 1, 2,... } and x is any variable.