Introduction

A detailed statement of what users (or clients or customers) of a program or program fragment expect it to do *and* what the implementers or developers of the code expect of its environment is called a *specification* for that code. Sometimes the user and developer of the code might happen to be the same person wearing different hats; however, it is best to think of them as independent, possibly with conflicting interests.

If the code being specified is sufficiently complex, several programmers might be involved in writing it and several other programmers might be involved in writing a program to use the code fragment. Furthermore, there might be several different implementations of a specification, and several different applications that use the implementations. A specification is essentially a *contract* among all these developers and users, stating exactly what must be agreed about the observable effects of executing the code and the environment in which it will be executing, and no more. The expectations of the users become obligations on the developers, and vice versa.

Normally, details of how the computational task is to be carried out would *not* be in a specification: the users shouldn't care, and implementers might then be prevented from using *other* implementation techniques. Similarly, a specification would normally *not* contain details of how applications are to use the code: the developers shouldn't care, and this might preclude *other* applications of the code being specified. To summarize, the specification for a program fragment should specify *what* it and its environment are expected to do but as little as possible about *how* or *why*.

The use of specifications is standard practice in every manufacturing and engineering field. For example, if you were considering the purchase of a particular model of printer for use with your home computer, you might want to obtain from the manufacturer or dealer its "technical specifications"; this document would include the following kinds of information:

- speed (pages per minute);

- resolution (dots per inch);
- memory (MB);
- input language (PCL, Postscript, etc.);
- duty cycle (maximum number of pages per month);
- power requirements (voltage and frequency ranges);
- power consumption (watts);
- operating systems supported;
- operating environment (acceptable temperature and humidity ranges);
- dimensions and weight;

and so on. Notice that some of these items impose obligations on the printer, whereas others impose obligations on the user of the printer.

In software engineering, the term *formal methods* is often used to describe development and validation techniques that are based on the use of logical and mathematical formalisms in specifications. Ideally, it should be possible to *construct* a software component systematically from its specification and to *verify* its compliance with the specification. This is not substantially different from any other branch of engineering, where it would be considered unprofessional *not* to use appropriate applied mathematics, such as circuit theory or statics. But in the relatively new field of software development, it is often claimed that using mathematical formalisms is unnecessary or impractical.

Unfortunately, conventional development methods are failing. Over 30% of enterprise software projects are canceled without being completed; 30% of the projects that are completed end up costing 150% to 200% of their original budget. Fewer than 10% of software projects in large companies are completed on-time and on-budget.

Defect rates in typical commercial software have been estimated at 10 to 17 per 1,000 lines of code. Studies at the University of Wisconsin have shown that over 40% of popular application programs on Windows operating systems may be made to crash or hang indefinitely simply by supplying them with randomly generated input data. Comparable failure rates have been observed for basic system utilities in some commercial UNIX–like operating systems.*

Developers and software vendors claim that eliminating software defects is impossible and that "bugs" are in any case only minor inconveniences; however, no one who has lost hours of work to a crashing word processor or has had to re-install their operating system is likely to agree. In some situations,

*The lowest failure rates were achieved by the open-source GNU utilities used on Linux systems.

defective programs are actually dangerous. A computer-controlled radiation-therapy device called the Therac-25 was involved in at least six incidents between 1985 and 1987 in which massive overdoses of radiation caused death or serious injury; the incidents have been attributed to software faults, exacerbated by inadequate system design, testing, and management procedures.

Even when software defects don't have such serious consequences, they are often very costly. Researchers at The Standish Group International estimate that software defects cost American companies about $100 billion annually in lost productivity and repairs. Here are some of the more spectacular failures of recent years.

- In 1990, the entire AT&T telephone network collapsed for nine hours because of a single programming error.
- In 1994, an error in the floating-point division algorithm implemented on Pentium processors cost Intel over $200 million.
- In 1996, the maiden flight of the *Ariane 5* space launcher ended in an explosion about 37 seconds after lift-off because an input conversion function, which had been used successfully with *Ariane 4*, could not cope with the larger values produced by the new version; the resulting unanticipated exception was handled by aborting execution of the inertial guidance code. The cost of this incident has been reported as being in the range of half a *billion* dollars. Ironically, the outputs of the conversion function were only being used for logging purposes and weren't needed for the actual flight.
- In 1999, an operating-system defect corrupted information in a crucial database, which caused the `eBay.com` web site to be inoperative for 22 hours.

Are *you* able to write code that correctly solves a simple programming problem? Try the following small exercise, using any programming language.

EXERCISE 1 Suppose `A` is an array of integers (but possibly with duplicated values) and that `n` is a nonnegative integer; write code to determine the number `nDist` of distinct values in `A` in the subscript range from `0` to `n-1`, inclusive. For example, if `n` = `6` and the first six components of `A` are `45`, `13`, `-15`, `13`, `13`, and `45`, respectively, `nDist` should be set to `3`.

Your code is not allowed to change `n` or `A`. The code should not be obviously inefficient; however, you may assume that `n` is sufficiently small that it is not worthwhile to sort (a copy of) the array segment initially.

Did you get the logic right *before* testing your code on a computer? Does your solution work correctly if `n` = 1? If `n` = 0? If all components of the array segment are the same? If all are different? Would you be willing to fly on an

airplane that is to be controlled by a program that uses your code? We will return to this programming problem in Section 3.6.

As consumers, managers, professional bodies, and regulatory agencies become more aware of the costs, dangers, and liabilities of poor-quality software, they will begin to demand that the software that they purchase or are responsible for be as reliable as other artifacts. Programmers will then have to become more professional than most commercial programmers are now. To keep their jobs, maintain their professional standing, and avoid malpractice suits, they will have to take responsibility for the quality of their products. It is increasingly a requirement on software for safety-critical systems such as aircraft, nuclear power-plant control, and medical equipment that it be supplied with detailed specifications, supported by formal or independent assessments of compliance.

Are you ready for a world in which you might have to take responsibility for the quality of a program? Consider the following amusing examples of this.

- In 1999, Ambrosia Software of Rochester, N.Y., announced that if any of their forthcoming software products subsequently required a bug-fix, their marketing manager would eat real insects at a trade show.*
- The government of China ordered the executive officers of the national airline to be on overnight flights on their airplanes on the night of December 31, 1999.

The program used to control the NASA space shuttles is a significant example of software whose development has been based on specifications and formal methods. A single defect in this program might result in the deaths of six astronauts and the loss of a multibillion dollar piece of equipment. It comes as no surprise that the group responsible for producing and maintaining this software have been obsessed with the correctness of their code.

As of March 1997, the program was some 420,000 lines long. The specifications for all parts of the program filled some 40,000 pages. To implement a change in the navigation software involving less than 2% of the code, some 2,500 pages of specifications were produced before a single line of the code was changed.

Their approach has been outstandingly successful. The developers found 85% of all coding errors before formal testing began, and 99.9% before delivery of the program to NASA. Only one defect has been discovered in each of the last three versions. In the last 11 versions of the program, the total defect count is only 17, an average of fewer than 0.004 defects per 1,000 lines of code.

It might be thought that this is an exceptional case whose success could not be approached in the "real world" of commercial software. But there are

*The press release did not say what would happen to the programmer responsible for the error.

commercially successful software houses that use the best available practices and regularly achieve defect rates of 0.03 to 0.05 per 1,000 lines of code. There is no technical or financial reason why at least this level of quality should not be demanded of all commercial and mission-critical software.

This book is an introduction to the use of software specifications. It describes basic formalisms suitable for specifying three kinds of code and practical techniques for systematic construction and verification of program components.

A small fragment of the C programming language is used for almost all the examples; some features of the `class` notation from C++, an object-oriented extension of C, are used to support information hiding in Part B. For the sake of readability and portability, a number of programming idioms that are specific to C (such as pointer arithmetic) will be avoided. If you have written programs in any similar language (JAVA, PASCAL, MODULA, ADA, TURING, etc.), you will have few difficulties reading the programs or constructing comparable programs.

Program 1

```
# include <stdio.h>
int main(void)
{ printf("Hello, world!\n");
  return 0;
}
```

The traditional first example of a C program is shown as Program 1. The effect of the program is to output the following line:

```
Hello, world!
```

The first line of the program has the effect of including the "header" file for the `stdio` (standard input-output) library, allowing the program to use the `printf` function defined in that library. The next line contains the "declarator" (heading) for a function (procedure, method) `main`, which is always called by the operating system to initiate program execution. The formal-parameter list `(void)` indicates that the function takes an *empty* argument list.

In C, a function-definition body is always a block (compound statement), which is a sequence of declarations and statements enclosed in curly braces `{...}`. The `printf` line is a call to a function that is defined in the `stdio` library and that may be used for formatted output. In this case, the argument consists of the literal string `Hello, world!`, terminated by the escape sequence `\n`, which generates suitable end-of-line control characters. The string is enclosed in double quotes, and the function call itself is terminated by a semicolon. The

Program 2

```
/* Test code to determine the number of distinct values in A[0:n-1] */

# include "specdef.i"

int main(void)
{
# define max 256     /* maximum number of entries, > 0         */
  typedef int Entry; /* type of entries, use == for equality   */
  int n;             /* number of entries                      */
  Entry A[max];      /* A[0:n-1] are the entries               */
  int nDist;         /* number of distinct entries in A[0:n-1] */

  printf("Enter n: ");
  if (scanf("%i", &n) != 1) error("input failure");
  if (n<0) error("n must be non-negative");
  if (n > max) error("n must be <= max");
  if (n>0)
  { int i;
    printf("Enter components of A[0:n-1], ");
    printf("separated by white space:\n");
    for (i=0; i<n; i++)
      if (scanf("%i", &A[i]) != 1) error("input failure");
  }

  ASSERT( 0 <= n <= max )

# include "mysolution.i"

  ASSERT( nDist == |A[0:n-1]| )

  printf("Number of distinct components in A[0:n-1] is ");
  printf("%i\n", nDist);
  return 0;
}
```

last line of the function body returns the value zero to the environment as an indication of successful completion.

Program 2 on page 6 is more useful; it may be used to test C solutions to Exercise 1 on page 3.

The first line is a comment describing what the program is for; in C, comments are bracketed by the character sequences `/*` and `*/`. The following `include` line has the effect of including the file `specdef.i`, which is listed here as Program 3. Note the use of double quotes `"..."` rather than angle brackets `<...>` to enclose the file name; this is to make the search for file `specdef.i` start in the current user folder/directory (rather than in system libraries).

The code in `specdef.i` includes headers for all the libraries likely to be used for ordinary programs and defines various types, macros, and functions assumed in the program examples in this book. *Our programs will always assume these headers and definitions, even if they are not shown explicitly.* The type defini-

Program 3

```
/* specdef.i:  headers and definitions for "Specifying Software" */

# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <ctype.h>
# include <limits.h>
# include <string.h>

# ifndef __cplusplus
typedef enum {false, true} bool;  /* not needed for C++ */
# endif

/* null macros:  */
# define ASSERT(P)
# define FACT(P)
# define INVAR(P)

void error(char msg[]) /* abort with stderr message msg */
{ fprintf(stderr, "Error: %s.\n", msg);
  exit(EXIT_FAILURE);
}
```

tion introduces an "enumerated" type `bool` of the two truth values, `false` and `true`; the compiler preprocessor is instructed to exclude this definition if the code is being processed by a C++ processor because the extended language predefines a `bool` type. The "null" macros `ASSERT(P)`, `FACT(P)`, and `INVAR(P)` allow for the use of special comment forms in programs; this usage will be explained in detail in subsequent chapters. The `error` function uses function `fprintf` to print out its error-message argument `msg` to the error stream and then aborts program execution by calling the `exit` function, using the constant `EXIT_FAILURE` as an indication of unsuccessful termination.

The body of the `main` function

- defines various constants and variables and the type name `Entry`;
- reads and verifies values for `n` and array components `A[0]` to `A[n-1]` using the `scanf` function from `stdio` (note the use of the ampersand "address" operator `&` on the input variables);
- executes the code found in a separate file `mysolution.i`, which has been `included` into the program by the preprocessor; and
- outputs the result.

The `ASSERT` lines are macro calls; these are essentially comments because the corresponding macro-definition body is empty. The significance of such "assertions" will be explained later.

Note the use of == in one of the assertions. Value equality tests are written this way in C programs; the = operator is used for the assignment operation. But when programs are discussed in the text, the usual $=$ symbol for mathematical equality will often be used, and similarly for other relational operators, such as $\neq$ and $\leq$.

Appendix A gives a compact reference manual for the programming language used in this book.

Additional Reading

The Therac-25 incidents are discussed in [LT93, Lev95, Neu95]. The *Ariane* incident is discussed in [JM97, Nus97]. The University of Wisconsin studies referred to are described in [MK+95, FM00]. The NASA space-shuttle control program is discussed in [Fis96]. The Usenet news group `comp.risks` carries discussions of errors and security loopholes in computer software.

Traditional attitudes to formal methods are criticized in [Hal90, BH95, LG97]. Real-world use of formal methods is described in [JS90, Hay92, Lin94, GCR94, CGR95, Har95, HB95, CW96, HB99, KH+00]. An overview of formal methods and introductions to a variety of formal specification languages may be found in [Win90]. Many additional bibliographical references on formal methods may be found here:

`http://www.comlab.ox.ac.uk/archive/formal-methods/pubs.html`

References

[BH95] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[CGR95] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions Software Engineering*, 21(2):90–8, 1995.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–43, 1996.

[Fis96] C. Fishman. They write the right stuff. *Fast Company*, 6:95–9 and 104–6, December 1996. Available here: `http://www.fastcompany.com/online/06/writestuff.html`.

[FM00] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proc. 4th USENIX Windows Systems Symposium*, Seattle, August 2000. Available here: `ftp://grilled.cs.wisc.edu/technical_papers`.

[GCR94] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 11(1):21–8, 1994.

[Hal90] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[Har95] J. M. Hart. Experience with logical code analysis in software maintenance. *Software Practice and Experience*, 25(11):1243–62, November 1995.

[Hay92] I. Hayes. *Specification Case Studies*. Prentice Hall International, 1992.

[HB95] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Prentice Hall International, 1995.

[HB99] M. G. Hinchey and J. P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. Springer Verlag, 1999.

[JM97] J.-M. Jézéquel and B. Meyer. Design by contract: The lessons of Ariane. *IEEE Computer*, 30(2):129–30, 1997.

[JS90] C. B. Jones and R. Shaw. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990. Available here: `ftp://ftp.ncl.ac.uk/pub/users/ncbj/cases.ps.gz`.

[KH+00] S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost-effective than testing? *IEEE Trans. Software Engineering*, 26(8):675–86, 2000.

[Lev95] N. G. Leveson. *Safeware: System Safety and Computers; A Guide to Preventing Accidents and Losses Caused by Technology*. Addison-Wesley, 1995.

[LG97] Luqi and J. A. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14:73–85, Jan. 1997.

[Lin94] R. C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–8, March 1994.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[MK+95] B. P. Miller, D. Koski, C. P Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical Report 1268, University of Wisconsin-Madison, Computer Sciences, April 1995. Available here: `ftp://grilled.cs.wisc.edu/technical_papers`.

[Neu95] P. G. Neumann. *Computer Related Risks*. ACM Press, 1995.

[Nus97] B. Nuseibeh. Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

[Win90] J. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.