Larch: Languages and Tools for Formal Specification

J.V. Guttag and J.J. Horning, with
S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing

January 19, 1993

Chapter 1

Specifications in Program Development

This book is about formal specification of programs and components of programs. We are interested in using specifications to help in the production and maintenance of high quality software.

We begin this chapter with a few remarks about programming and the role of abstraction. We then move on to discuss how specifications fit into the picture.

1.1 Programming with abstractions

Building a software system is almost entirely a design activity. Unfortunately, software is usually designed badly or barely designed at all. A symptom of negligence during design is the number of software projects that are seriously behind schedule, despite having had design phases that were "completed" right on schedule [10]. In practice, design is the phase of a software project that is declared "complete" when circumstances require it. Part of the problem is that there are few objective criteria for evaluating the quality and completeness of designs. Another part is the elapsed time between producing a design and getting feedback from the implementation process.

This book describes how formal specifications can be used effectively to structure and control the design process and to document the results.

The key to structuring and controlling the design process is, as Machiavelli said, "*Divide et impera*." Regrettably, he was not clear about how to apply this stratagem to software development.

Two primary tools for dividing a problem are *decomposition* and *abstraction*. A good decomposition factors a problem into subproblems that:

- are all at the same level of detail,
- can be solved independently, and
- have solutions that can be combined to solve the original problem.

```
int sqrt(int x) {
  requires x ≥ 0;
  modifies nothing;
  ensures ∀ i: int
    ( abs(x - (result*result)) ≤ abs(x - (i*i)) );
}
```

FIGURE 1.1. A specification of an integer square Root procedure

The last criterion is the hardest to satisfy. This is where abstraction comes in. Abstraction involves ignoring details that are irrelevant for some purpose. It facilitates decomposition by making it possible to focus temporarily on simpler problems.

Consider, for example, the problem of designing a program to compile a source language, say Modula-3, to a target language, say Alpha machine code. Much of the compiler can be designed without paying attention to many of the details of either Modula-3 or the Alpha architecture. One might well begin by abstracting to the problem of compiling a source language with a deterministic context-free grammar to a reduced instruction (RISC) set target language. One might then choose to model the compiler's design on the design of other compilers that solve the same abstract problem, e.g., to decompose the problem into the separate problems of writing a scanner, a parser, a static semantic checker, and several code generation and optimization phases.

This paradigm of abstracting and then decomposing is typical of the program design process. Two important abstraction mechanisms are used: abstraction by parameterization and abstraction by specification.

Abstraction by parameterization allows a single program text to represent a potentially infinite set of computations or types. For example, the C code

```
int twice(int x) {return x + x;}
```

denotes a procedure that can be used to double any integer.

Abstraction by specification allows a single text to represent a potentially infinite set of programs. For example, the specification in Figure 1.1 describes any procedure that, given an appropriate argument, computes an integer approximation to its square root. Notice that it specifies the required result, not any particular algorithm for achieving it. Notice also that it does not describe the result completely. For example, it does not constrain the result to be positive.

For the most part, software design is the process of inventing and combining abstractions and planning their implementation.

There are several reasons why it is better to think about combining abstractions than to think about combining their implementations:

- Abstractions are easier to understand than implementations, so combining abstractions is less work.
- Relying only on properties of the abstractions makes software easier to maintain, because it is clear what properties must be preserved when an implementation is changed.
- Because an abstraction can have several implementations with different performance properties, it can be used in various contexts with different performance requirements. Any implementation can be replaced by another during performance tuning without affecting correctness.

The key to good software design is inventing appropriate abstractions around which to structure the software. Bad programmers typically don't even try to invent abstractions. Mediocre programmers invent abstractions sufficient to solve the current problem. Great programmers invent elegant abstractions that get used again and again.

1.2 Finding abstractions

Structure is sometimes identified with hierarchy; hierarchical decomposition is sometimes preached as the only "structured" programming method. The problem with hierarchical decomposition is that, as the hierarchy gets deeper, it leads to highly specialized components that assume a great deal of context. This decreases the likelihood that components will be useful elsewhere—either in the current system or in software that is built later. A relatively flat structure usually encourages more reuse.

Important boundaries in the software should correspond to stable boundaries in the problem domain. Such correspondence makes it more likely that when customers ask for a small change in the observed behavior of the software, the change can be accomplished by a small change to the implementation. Stable boundaries in the problem domain frequently involve data types, rather than individual operations, because the kinds of objects that long-lived software manipulates tend to change more slowly than the operations performed on those objects. This leads to a style of programming in which data abstraction plays a prominent role.

A *data type* (data abstraction) is best thought of as a collection of related operations that manipulate a collection of related values [68]. For example, one should think of the type `integer` as providing operations, such as `0` and +, rather than as an array of 32 (or perhaps 64) bits, whose high-order bit is interpreted as its sign. Similarly, one should think of the type `bond` as a collection of operations such as `get_coupon_rate` and `get_current_yield` rather than as a record containing various fields.

An *abstract type* is a type that is presented to a client in terms of its specification, rather than its implementation. To implement an abstract type, one selects a *representation* (i.e., a storage structure and an interpretation that says how values of the type are represented) and implements the type's operations in terms of that representation. Clients of an abstract type invoke its operations, rather than directly accessing its representation. When the representation is changed, programs that use the type may have to be recompiled, but they needn't be rewritten.[1]

Even in languages, such as C, that provide no direct support for abstract types, there is a style of programming in which abstract types play a prominent role. Programmers rely on conventions to ensure that the implementation of an abstract type can be changed without affecting the correctness of software that uses the abstract type. The key restriction is that programs never directly access the representation of an abstract value. All access is through the operations (procedures and functions) provided in its interface.

1.3 The many roles of specification

Abstractions are intangible. But they must somehow be captured and communicated. Specification gives us a way to say what an abstraction is, independent of any of its implementations. Writing specifications can serve to clarify and deepen designers' understanding of whatever they are specifying, by focusing attention on possible inconsistencies, lacunae, and ambiguities.

Once written, specifications are helpful to implementors, testers, and

[1]For a more comprehensive discussion of the role of data abstraction in programming, see [63].

maintainers. Specifications provide "logical firewalls" by documenting mutual obligations. Implementors are to write software that meets its specification. Clients, i.e., writers of programs that use the software interface, are to rely only on properties of the software that are guaranteed by its specification.

During module testing and quality assurance, specifications provide information that can be used to generate test data, build stubs, and analyze information flow. During system integration, specifications reduce the number and severity of interface problems by reducing the number of implicit assumptions. Finally, specifications aid in maintenance by recording the properties that must be preserved and by delimiting the changes that might affect clients.

All of these virtues can be attributed to the information hiding provided by specifications. Specification makes it possible to completely hide the implementation of an abstraction from its clients, and to completely hide the uses made by clients from the implementor [70].

1.4 Styles of specification

A good specification should be tight enough to rule out implementations that are not acceptable. It should also be loose enough to allow the most desirable (i.e., efficient and elegant) implementations. A specification that fails to rule out undesired "solutions" is not sufficiently constraining; one that places unnecessary constraints on implementations is guilty of *implementation bias*.

A *definitional specification* explicitly lists properties that implementations must exhibit. The specification in Figure 1.1 is definitional. An *operational specification* gives one recipe that has the required properties, instead of describing them directly. Figure 1.2 contains an operational specification of a square root procedure. It looks suspiciously like a program—it defines a function by showing how to compute it. In fact, every program can be viewed as a specification. The converse is not true: many specifications are not programs. Programs have to be executable, but specifications don't. This freedom can often be exploited to make specifications simpler and clearer.

There are strong arguments in favor of both the operational and definitional styles of specification. The strength of operational specification lies in its similarity to programming. This reduces the time required for programmers to learn to use specifications. Some operational specifications

```
int sqrt(int x)
  requires x ≥ 0
  effects
    i = 0;
    while i*i < x
      i = i + 1 end
    if abs(i*i - x) > abs((i - 1) * (i - 1) - x)
      then return i - 1
      else return i
```

FIGURE 1.2. An Operational Specification of Integer Square Root

are directly executable. By executing specifications as "rapid prototypes," specifiers and their clients can get quick feedback about the software system being specified.

On the other hand, definitional specifications are not bound by the constraint of constructivity. They are often shorter and clearer than operational specifications. They are also easier to modularize, because properties can be stated separately and then combined. Because definitional specifications are so different from programs, they provide a distinct viewpoint on systems that is frequently helpful.

It is often difficult to determine from an operational specification which properties are necessary parts of the thing being specified and which are unimportant. The specification in Figure 1.2, for example, allows fewer implementations than the specification in Figure 1.1. An implementation is certainly not obliged to use the simple, but horribly inefficient, specification algorithm, but it must compute the same result, and therefore must not return a negative number. This constraint might be essential in some contexts and insignificant in others. Figure 1.2 does not say, and cannot easily be modified to say, whether the sign of the result matters. Figure 1.1, on the other hand, can easily be strengthened to specify the sign if that is important.

1.5 Formal specifications

The specifications in this book are written in formal specification languages. A formal specification language provides:

- a *syntactic domain*—the notation in which the specifications are written,
- a *semantic domain*—a universe of things that may be specified, and
- a *satisfaction relation* saying which things in the semantic domain satisfy (implement) which specifications in the syntactic domain.

We use formal languages because it seems to be the easiest way to write specifications that are simultaneously precise, clear, and concise. This is hardly surprising. It is no accident that such diverse activities as chemistry, chess, knitting, and music all have their own formal notations.

Mistakes from many sources will crop up in specifications, just as they do in programs. A great advantage of formal specification is that tools can be used to help detect and isolate many of these mistakes. Anyone who has used a strongly typed programming language knows that even something as simple as a syntax and type checker is invaluable. Comparable checking and diagnosis of formal specifications is easy and worthwhile, but we can do even better. Various kinds of formal specifications can be checked more thoroughly by tools that help explore the consequences of design decisions, detect logical inconsistencies, simulate execution, execute symbolically, prove the correctness of implementation steps (refinements), etc.

Are formal specifications too "mathematical" to be used by typical programmers? No. Anyone who can learn to read and write programs can learn to read and write formal specifications. After all, each programming language is a formal language.