Chapter 1

Introduction

1.1 What and why

This book deals with formal specification of complex software systems. Rather than advocating a single specification method, several possibilities are described, and the relations among these approaches are examined. Before describing the methods, some clarifications are appropriate to explain what is meant by specification and the other terms we shall be using throughout the book. Informally, a specification is a description of assumptions and requirements about computations of a software system. What is required should be well understood, but there can be aspects of the system about which no requirements are given. In other words, the specification should be unambiguous, but it need not be complete.

Formal specifications can either be used (1) to help avoid misunderstanding between the client/management/marketing level and the design level of a software development, or (2) to encourage precise interfaces within the design level and use of specifications as the basis of verification or testing of implementations (code). The first is known as Requirements Specification, and the second is known as Design Specification. Ideally, we would prefer a specification method that supports both uses. However, in practice, industrially robust specification techniques that have associated tools put emphasis on one or the other of these aspects. This will be explained in greater detail later.
Although the way in which requirements can be expressed is a key subject, very little will be said about how to determine these requirements, especially for Requirements Specification. This subject is treated in the requirements analysis stage of software engineering. It can include techniques for interviewing clients and eliciting their needs. Aspects of Operations Research and Business Administration can also be involved. Although significant research is underway in these areas, they are more a part of the social sciences than computer science. Unfortunately, the nature of human interactions is poorly understood even by experts in those areas. This book will therefore treat the intermediate area between vague client intuitions of their needs, and implemented code of “the systems people.” However, the criteria for evaluating whether a specification method is practical are discussed below, and these are influenced by the aid provided in identifying the intentions of the clients.

It is natural to ask why specification languages are needed at all. Why do we need to impose another formalism between the intuition of the client and the code of the implementor? The reasons all relate to the increased complexity of modern software systems, and to the widespread dissatisfaction with the present state of affairs in complex system development. It is not even clear what we want from such systems. What exactly is expected of an operating system, or an elevator controller, or a telephone switch, or an optimizing compiler, or a window management system? The specification can be viewed as a contract between the client and the implementor. In fact, specifications are sometimes used as part of actual legal contracts. This application is one reason that many large companies with government contracts in, e.g., aerospace, defense, or financial services, are very eager to apply formal specification methods (even though they only rarely do so, for reasons which will be analyzed later). Disagreements about whether a software project satisfies a (written) contract are both common and difficult to resolve as long as the requirements are given only in vague natural language documents.

If the vaguely formulated user intentions are somehow “above” the level of the formal specifications, then the program or system implementation is below it. The question arises of the difference between a specification language and a programming language. In early work
on this subject, the distinction was easy: a programming language expresses how to execute (i.e., an algorithm), while a specification language expresses what (i.e., predicates or relations among values). There is still a good deal of truth in this view. A program to sort an array using heapsort appears very different from a specification of sorting, stating merely that the result is a permutation of the input with increasing values.

However, today this distinction is blurred: in one direction, a programming language such as Prolog can be viewed either as a statement of relations among data items, or as a program to determine results using unification among clauses and left-most depth-first search as a “computing engine” on the given logical clauses. On the other hand, the requirements in a specification can be viewed as defining a collection of possible computations, just as a programming language. Some specification languages, especially for distributed systems, really are used to describe algorithms, using high-level primitives of the language. Thus such languages resemble unimplemented programming languages.

An important difference which does remain: the description of the collection of computations does not have to include a description of an efficient (or even reasonable) way of computing them, even though it could include a requirement of efficiency. In fact, a specification could describe requirements which cannot be computed at all. As a case in point, it is not difficult to describe the halting problem: “given the text of a program as input, determine whether that program must always terminate, for any possible input program”. However, there is no algorithm that can satisfy this specification, since the problem is known to be undecidable. Such a requirement can be expressed in many specification languages, but not in any programming language.

1.2 What is a computation?

A specification limits what may occur during computations of a program or system. Thus we first need to consider what constitutes a computation. Several alternative views are reasonable. Among these, a computation can be seen formally as a collection of events with some partial ordering among them, as a sequence of events, or as a sequence
of states of the system. In an *event* or *action*-based approach, the specification indicates which combinations of actions are possible, or, for example, when two sequences of actions can be considered equivalent. In a *state*-based approach, predicates that should hold in some of the states are defined.

In both cases, the specification generally relates to key actions or to those states that are significant, often because they are part of the interface of the system being specified. That is, the specified entities are externally visible to users of the system, and not merely restrictions on internal actions or states.

In this book we do not want to limit our view of computation arbitrarily. As an example, in recent years increasing numbers of systems are intended to execute forever. Such systems are expected to react to input values as specified, and to continue executing, unless a termination condition occurs. Therefore we should not build into the specification method an assumption that computations are bounded, or even necessarily finite. Both terminating and non-terminating systems are today commonly found and need to be specified.

The term *reactive* has been used to describe such general non-terminating systems, including electronic mail, databases, operating systems, airline reservation systems, and automatic teller banking systems. In fact, many modern computer systems are reactive, rather than simply computing a simple function.

Even systems that take an input and compute a single result, and thus could not be called reactive, are increasingly complex. Many such systems, such as compilers, or hardware verification tools, have aspects in their specification that are difficult to capture in standard mathematical notations. The specification methods we consider can all treat reactive and/or complex systems to some degree.

Most of the properties we will be describing can be divided into two broad categories, *safety* and *liveness*. Intuitively, safety properties describe what is to be maintained during the computation, while liveness properties capture what must be achieved. Safety properties imply that forbidden events or situations cannot occur. Stated more positively, the safety properties describe possible events or states. On the other hand, only liveness properties guarantee that particular events or states must occur during some or all of the computations of the system, ensuring
that progress is made during execution of the system. As the various methods are presented we shall indicate whether the method allows expressing safety properties, liveness properties, or both. Temporal logic, in particular, will be shown to be appropriate for expressing general liveness properties, and for precisely defining these terms.

Real-time and fault tolerance are two other families of properties that often need special techniques, both for specification and implementation. These have only recently been introduced to formal specification methods, and the techniques for doing so are less established than for safety and liveness. Later in the book these topics are surveyed for a few specification methods.

1.3 Uses of Specifications

Both the advantages and disadvantages of formal specification will be considered in much greater detail in later chapters, but here we still can list a few of the advantages, even when contractual obligations are not involved.

1. Correctness. Theoretical computer scientists were motivated to consider specifications to allow proving the correctness of programs. Obviously, a program is only correct relative to some requirements, and until those can be expressed, there is no formal possibility of showing correctness. For systems in essential applications or with extensive use, it is possible to formally verify at least some aspects of the specification through static logical analysis of the software code. There are several automatic tools to aid in this, although most are still experimental, and verification has proven to be difficult. The techniques show most practical promise for finite-state algorithms that are high-level hardware designs or communication protocols. Even when full verification is not feasible because of a lack of automatic tools, time, and resources, taking issues of formal correctness into account during development can be worthwhile, and lead to eliminating many potential errors before they are buried deep within the code of an implementation.
2. **Coverage.** The need to write a specification in a given formalism often ensures that all components of a problem are related to at an early stage. This has been called “specification as a checklist,” and has been cited as beneficial in many examples. Overlooking important requirements until late in the design or even implementation stages can lead to costly redesigns and resultant slippage in scheduling.

3. **Consistency.** Even without full correctness proofs, specifications often can aid in discovering unreasonable assumptions or contradictory requirements at an early stage, before any investment in implementation has been made. For this reason it is often useful to have redundancy in specifications, allowing internal consistency checks to be made.

4. **Reusability.** When hundreds or even thousands of software components are gathered into libraries that facilitate reuse in a variety of contexts, it is difficult to know the purpose of each component. A software tool for manipulating these components would surely benefit from having formal specifications, capable of being analyzed automatically, so that module specifications and user requirements could be matched. The “software factory” approach advocated by Japanese industry is based on such libraries. If precise specifications of the modules existed, it would make sense to invest the effort of verifying the library modules relative to their specifications, thereby increasing the reliability of resultant systems. This would be in addition to using the specifications to identify appropriate modules for new applications.

5. **Documentation.** A formal specification can serve as an ongoing reference document to help in clarifying the intention of subtle points in the system, and the interactions of separately specified modules. It can also serve as the basis for more informal user and system manual descriptions of a system. This does require updating the specification along with the code as modifications are made to the system, but can yield significant savings in subsequent understanding.
Consistency and Coverage relate to the use of specifications in Requirements specification, to avoid misunderstanding between clients and implementers. On the other hand, Correctness and Reuse are mainly relevant for Design specification, i.e., interfaces among design modules and between design and implementation, while Documentation is relevant to both aspects.

1.4 General Evaluation Criteria

There are tens, in fact hundreds, of notations for describing the requirements of software systems or programs. Thus it is not possible to adequately treat all of them, or even all of those that are actively being used or developed. Since the field is changing rapidly, it is clear that new specification languages and methods will appear in the coming years. What can be done is to understand the general types of specification methods in widespread use, and to consider in greater detail an example of each type. Once the possible parameters in a specification method are understood, and an example of each type has been demonstrated, it should become easier for the reader to place a new specification method into a wider context, and to understand why various language design decisions were made.

There are numerous criteria for evaluating the quality of a specification method, or of a particular instance of a specification. All of the criteria help in providing Documentation and in encouraging Reuse. The other uses of specifications are emphasized to different degrees by the different methods, and thus are included in some of the more specific criteria given below. Among other desirable features, a specification should be:

1. **Precise.** The semantics of the specification language must be unambiguous in order to avoid misunderstanding between those making the specification and those who use it. For example, if a priority list is given in the order (with numbering) (1), (2), (3), and followed by the statement “the highest priority error is the one displayed...” should error (1) or (3) be displayed if both occur?
2. **Accessible.** The specification should be easy to understand and explain to others. In particular, it should not be accessible only to trained experts in, say, category theory, and should be able to incorporate terminology common to a particular domain.

3. **Expressive** The desired properties natural to a problem domain should be easily expressible in the specification, for many types of systems. This means that the method is widely applicable.

4. **Modular.** The specification should be expressible in stages, without having to go back and modify previous parts of the specification when adding new requirements. The method should be extendible in order to incorporate new aspects as the need for them arises, without fundamentally changing the specification method.

5. **Hierarchical.** A specification should be expressed, and then examined, at various levels of detail. Sometimes, high-level requirements are needed, while in other situations, more detail is required. Examination and development of all levels should be facilitated.

6. **Analyzable.** A methodology, or, preferably, an associated tool is available to check the internal consistency or implications of the specification, or to identify aspects that seem to be unclear, such as the interrelations among seemingly unrelated operations. Thus both Consistency and Coverage are encouraged by such tools.

7. **Executable.** A prototyping or simulation capacity should be associated with at least some aspects of the specification. That is, the specification method has related tools that allow (inefficient) computation for sample input. These should demonstrate possible behaviors of the system being specified, and help clients determine whether the intended requirements have been expressed. Note that some aspects of the specification could be impossible to simulate.

8. **Provable.** Tools or methodology are developed to enable proving correctness, i.e., that the specification is satisfied by the imple-
1.4. GENERAL EVALUATION CRITERIA

Implementation, for every possible input. This is usually most practical for the aspects of the problem expressible as a finite collection of possible states.

9. **Refinable.** The method provides a clear path for refining a specification through gradual stages to reach an implementation, or at least for clearly connecting the implementation level to the design specification. This is clearly related to the previous criterion of being Provable, but emphasizes more the aid that the specification can provide in reaching an implementation in a standard programming language.

10. **Minimal.** The specification should not commit to more than is needed. Otherwise, the possibilities for implementing the requirements are unnecessarily restricted, preventing optimal use of resources.

As described earlier, the reasons for specifications differ for Requirements Specification and Design Specification. Requirements Specifications are used mainly to provide feedback and ensure proper coverage and interaction among different components at an early stage. Design Specifications are to be verified with respect to the Requirements, and to provide a standard for verification of the implementation, including aid for testing.

Most of the criteria above are relevant both for Requirements and for Design, but sometimes with differing emphases. A method can be refinable within its own notation, to provide an easy connection from Requirements to Design, but not allow easy concrete implementation, or, on the other hand, be easy to implement, but not allow easy expression of high level Requirements, separate from implementation details.

Unfortunately, even if specification languages existed that satisfied all of the above criteria, it is not immediate that they would immediately be adopted for large-scale industrial use. There are numerous special difficulties, some of them sociological, that make widespread introduction of specifications methods difficult. One difficulty is that industrial systems are rarely developed from scratch. It is far more common to upgrade an existing system, or at least to build the new
system as a collection of modifications of older ones, sometimes incorporating modified code from earlier systems, especially when an entire line of related systems is developed. Since the older system made no use of formal specifications, it is difficult to start incorporating the new approach for only part of the system.

Similarly, new specification techniques are usually not properly integrated with existing company standards for software development, documentation, and testing. Often there are test-beds and other in-house tools that do not interact with specification methods.

However, the primary difficulty in introducing specification into production projects is that the development team is operating under strict time and resource constraints. The specification is not considered a goal in itself, but only a tool to help in improving the quality and delivery time of a system project. Thus the designers are not willing to invest the effort of learning a new notation, development methodology, and associated tools in parallel to writing a new system. The new specification technique requires skills that may be different from those emphasized in the experience of the development team.

This barrier can be overcome only by committing sufficient resources to such a change in development methodology. Learning a new specification method is a task that takes time and effort. The first project using specifications may not benefit from reduced development time because of this learning overhead. The economic benefits of using formal specifications should be considered over a longer term, aggregated over several projects and versions of systems. In that case, I would claim that the benefits can be real and significant. The short-term difficulties should not be allowed to prevent introducing techniques that can bring the advantages noted above.

1.5 Varieties of specification methods

In this section several categories of notations for formal specification are briefly outlined. Some of the methods are not appropriate for the kinds of system to be treated here, either because they only treat very simple cases, or because they are too close to what would traditionally be called a programming language. These are not amplified upon later
in the book. For the other methods, separate chapters are devoted to
an example from each category.

In the later part of the book we will consider a variety of problem-
atic requirements that arise in modern distributed, reactive systems,
and how they are treated by various methods. In particular require-
ments for real-time and for fault tolerance are expressed with some of
the methods presented earlier. The development of appropriate spec-
fications for such requirements is an active area of research, and no
standard notation yet exists. Therefore, these chapters are more spec-
ulative than the rest of the book, and are intended to point out the
issues involved.

One of the main conclusions drawn from this investigation is that
there is no single "winner" among the specification languages. Some
types of requirements are especially easily and naturally handled by
one method, but other requirements are especially awkward to express
for that same method, and are most natural in another notation. In
choosing which languages to highlight from each type, one of the main
considerations was on the level of user interface, and supplementary
tools that exist for the language. The notations chosen for emphasis in
this book seem especially user-friendly, or deal with some of the difficult
issues in a particularly clean way. However, it should be understood
that other specification languages may have advantages not in those
highlighted here, and that each of the actively used ones has enthusiasts
who consider it the notation of choice for specifications.

1.5.1 Input-output specifications

We first consider some of the notation to specify programs used origi-
nally in papers on proving correctness of programs[15]. It is assumed
that a single module is being considered, and that some underlying
mathematical notation is available. The most commonly used notation
is first-order predicate calculus, augmented with standard mathemati-
cal notation. A program or module is specified by two predicates. One
gives the assumptions about legal input values, while the other gives a
relation between initial and final values of computations. This approach
clearly can be impractical for reactive systems, or for requirements that
do not correspond directly to the usual mathematical symbols. Nev-
ertheless, it is important to understand this method, since it serves as the basis of several other approaches, and since it still is an effective specification method for some classes of systems.

1.5.2 Algebraic specification

Although simple input/output assertions were adequate for toy programs, as seen in early works on program correctness, they are inadequate to describe most system specifications. One fundamental problem is that the assertions use standard mathematical notation and first-order predicate calculus to describe the state. But from the 1960's onward, it became clear that methods of organizing data are essential to efficient execution of programs, and that these methods use data structures that are not captured by common mathematical operations. For example, a stack has operations of putting an element into the stack, and removing or examining the last element put in the stack that has not yet been removed. These operations simply are not primitives in accepted mathematical notation, although they can be expressed indirectly in a variety of ways.

Algebraic specification methods emphasize the relations among the operations of a data structure. In fact, such methods were originally developed for the description of a single data structure such as a stack or an array. Gradually, it was realized that almost any reactive system can be viewed as a generalized data structure responding to user operations. Thus algebraic specification can be viewed as a general class of specification methods, although the emphasis is still on sequential systems.

One of the most user-friendly algebraic specification methods is the Larch approach of John Guttag, Jim Horning, and Jeanette Wing, to be presented in a later chapter.

1.5.3 Z and a set-based approach

Another base for specification can be seen as set theory. As every mathematician knows, all of mathematics, and indirectly all of science, can be expressed through set-theory. This theoretical view of course is not at all reassuring on how a set-based approach would prove to be in
1.5. VARIETIES OF SPECIFICATION METHODS

a practical setting. How difficult is it to express the real requirements of industrial systems using such a notation? Can anyone besides a set-theorist do it? The somewhat surprising answer is that the Z formalism provides a widely used and expressive set-based specification method.

Z is an example of a model-based theory, where a particular model — in this case set theory — is assumed known. That is, notation for sets and set operations are built into the language and assumed understood by the user, and then are used to define new operations and desired properties.

Special care has been taken to make the resultant system modular, in that it is built up piece-by-piece. The refinement of a high-level specification into lower-level ones is also emphasized. The developers of Z, primarily at the Oxford Laboratory for Programming,[33, 32], have demonstrated that their method can be used for complex systems, such as telephone exchanges, databases, and compilers.

1.5.4 Statecharts and state machines

Yet another category of specification methods breaks out of the conventional mold — of lines of symbols — into a two-dimensional graphical presentation. In this approach, the transitions from state to state are indicated graphically. A hierarchical structure is encouraged, where a top level view hides most of the detail, while a zoom facility allows examining and developing more detailed views of the system requirements. The visual notation encourages a global picture of the system structure and internal components, while the zoom facilities allow hiding extraneous details. A simulation facility is used to check that the specification conforms to the intuitive expectations.

The first to advocate such a graphical approach was David Harel[14], who proposed the Statechart notation, later developed into the StateMate specification system.

Several specification methods resemble statecharts in dealing with transition systems, but are text-based with emphasis on communication among modules in asynchronous distributed environments. These have been called state-machine approaches. Among the specification methods in this category are the Input/Output automata of Lynch, Tuttle, and others [24, 25, 23], and the state-machine advocated by
Lamport[20, 22]. This approach has been used to describe subtle interactions among communication subsystems, including fault-tolerance and real-time.

1.5.5 Temporal logic

Next, we turn to the class of specification methods best suited to describing general liveness properties. Temporal logic is a logic with special symbols (called modalities) that relate to sequences of states obtained by a global, atomic view of the semantics of a program. As opposed to the input/output assertions seen above, it does not relate to specific initial or final states, but rather to the possible sequences of states that can occur during execution. Many requirements that are natural for reactive systems are easy to express in temporal logic. Among these are statements such as “every request is eventually answered” or “P is true until Q becomes true”.

Temporal logic was introduced in the context of specifications for programs by Amir Pnueli[31]. It is the subject of a recent book by Pnueli and Manna [26] and has been advocated strongly by Leslie Lamport[21], among others. Numerous varieties of temporal logics have been suggested, and we shall survey some of the possibilities.

1.5.6 Process algebras

Several specification methods emphasize building distributed systems by composing lower level operations in what is known as a process algebra. CCS and CSP (by Robin Milner[28] and C.A.R. Hoare[16], respectively) are two (of many) such algebraic description languages for distributed systems. A more industrially oriented language with some similar ideas is LOTOS[3]. All of these are used to describe particular algorithms, and in that sense can be viewed as (usually unimplemented) programming notations as well as specification methods. They are also valuable in providing insight into the nature of distributed computing.

We will be considering LOTOS in a later chapter, as an example of this approach. In addition to embodying ideas of process algebra, LOTOS allows expressing multiparty interactions. These are high-level atomic actions with a group of processes that come together,
synchronize and exchange values, and then continue separately. A design methodology that then relaxes the synchrony among the processes seen in these operations is suggested. Such a view has been the basis for additional specification methods that are not process algebras. For example, they can also be seen in the work on Raddle, Verdi, and IP from the MCC Software Technology Program[9, 10], in the action systems and the Disco language of Reino Kurki-Suonio[17, 19], and in several proposals for multicast primitives for distributed systems.

1.5.7 Methods not taken

Among the other specification methods not considered in separate chapters are Petri nets, VDM, SDL, and Unity. In the concluding chapter, several of these methods are discussed and compared with those we have been able to treat in more detail.

Petri nets [30] are really a collection of related graphic specification methods for distributed systems. They have been the subject of intense research over a period of years. The Petri community has long advocated viewing an execution as a partial ordering among events in various processes. Some of the graphical ideas of Statecharts can be traced to that model, as well as the partial order view seen in LOTOS. Multiprocess interactions are naturally supported by the notation.

VDM is yet another extensively investigated model [2, 18]. In VDM, a system is viewed as an abstract machine, with various levels of detail, and an operational definition is given of the system. It can be viewed as a model-based approach, and in recent years relations with Z have been emphasized through joint conferences.

Unity is a notation developed by K. Mani Chandy and J. Misra [4] to capture the common denominator of parallel algorithms appropriate for a variety of computational models. The idea is that a common core algorithm can be transformed into one for systolic arrays, asynchronous message-passing systems, synchronous shared memory, etc. In the Unity view, nondeterminism is more basic than parallelism, and the division into processes is a low-level implementation decision. In addition to the Unity notation, an associated temporal logic is used to show abstract properties of programs written in the notation.

Of course, there are also many informal methods, that combine
an outline of a system with natural language descriptions of parts of the functionality. These are not treated here because of the inherent ambiguity and unclear semantics that inevitably result by introducing natural language descriptions.
Chapter 2

Input/output assertions and logic

2.1 Basic assumptions

As noted in the Introduction, to show that a program is correct we first need to state precisely what is required from it, i.e., its specification. Formal specifications known as input/output assertions were first used as the prerequisite of such proofs. The simplifying assumptions made in the input/output approach to specification, at least in its original context, are considerable: that a single module is being considered, and that this module is intended to compute a mathematical function of its input.

Under these assumptions, it is sufficient to express which inputs are acceptable, and for each legal input, what is the expected output. Semantically, an input/output specification consists of a collection of possible values for the initial state and a collection of pairs of values that represent acceptable initial and final states. In other words, for each execution sequence, there are only two states of interest that need to be specified: the initial and the final state.

For specifications, we must consider how such collections of values and pairs can be reasonably expressed. In the approach here, logical predicates will be used to give these collections in a precise, compact form. In practice, within the predicates some underlying mathematical
notation is used freely without a formal definition. Most commonly, the specification language is first-order predicate calculus, augmented with what could be called standard mathematics: the usual accepted notations of functions and standard, universally understood symbols. Thus `+' and `*' denote addition and multiplication of two values, \( n! \) denotes the factorial of \( n \) (i.e., \( n \times (n - 1) \times \ldots \times 2 \times 1 \)), and \( \Sigma \) denotes summation. Even the ... notation itself (denoting “from-to”) should be included in this category of accepted mathematical notation. Aspects that are difficult to define precisely, but which everyone is assumed to understand, are given as undefined predicates, especially for standard data domains. Thus \( \text{integer}(x) \) is used to denote that \( x \) is an integer.

## 2.2 Mathematical logic for assertions

Although logic is a valuable tool for many aspects of computing, here readers are assumed only to have some familiarity with basic logic notation and terminology. Below the particular symbols used in the book are defined rather informally.

As noted above, we shall assume that expressions in various domains are written using standard variables and connectives such as function application, addition, and logical inequalities between expressions. The mathematical notations are combined into expressions and inequalities about the state of the program.

The logical connectives used are:

- \( p \lor q \) (disjunction) is \textit{true} if and only if either \( p \) or \( q \) are \textit{true}.
- \( p \land q \) (conjunction) is \textit{true} if and only if both \( p \) and \( q \) are \textit{true}.
- \( \neg p \) (negation) is \textit{true} if and only if \( p \) is \textit{false}.
- \( p \rightarrow q \) (implication) is \textit{true} if and only if \( q \) or \( \neg p \) are \textit{true}.

The arguments of the connectives are called \textit{clauses}, so that implication has a left clause and a right clause. A convenient shorthand for inequalities is \( a \leq x \leq b \), which stands for \( a \leq x \land x \leq b \).

Two standard quantifiers are also defined as usual in mathematical logic:
2.2. **MATHEMATICAL LOGIC FOR ASSERTIONS**

- \( \forall x. p(x) \) (universal quantification) means for every value from the domain of \( x \), \( p \) is *true* when that value is substituted in place of every occurrence of \( x \) in \( p \). It can be seen as a generalized (possibly infinite) conjunction over the clauses obtained by substituting each possible value for \( x \).

- \( \exists x. p(x) \) (existential quantification) means that there exists a value from the domain of \( x \), for which the substitution of that value in place of \( x \) in \( p \) yields a *true* predicate. Existential quantification can thus be seen as a generalized disjunction.

There is a crucial difference between the use of variables in logic and in programs: in logic, a ‘name’ is used to represent a fixed value, while in programs, a variable is usually a dynamically changing object with different values in different states. Quantification over a domain in logic merely represents a set of possible values, while a variable in a program is an entirely independent object. In assertions about programs, a free (unquantified) variable is used to refer to a program variable, while a quantified variable is a place-holder for any one of the possible values in the domain of the quantified variable. Thus \( \forall i. x \neq i \) means that the value of the variable \( x \) is not in the domain over which \( i \) is defined.

When quantifiers are used, it is common to restrict the domain of the quantifier. For universal quantification, this can be done using implication, as in:

\[ \forall x. 1 \leq x \leq 10 \rightarrow y \neq x^2 \]

Note that the quantification is over *all* possible values of \( x \); for those not in the range between 1 and 10, the assertion is trivially *true*, because the left side of the implication is *false*, while for all values in the range the program variable \( y \) must be different from \( x^2 \) (at least if the assertion is intended to be *true*).

On the other hand, for existential quantification restricting the domain of the bound variable requires conjunction:

\[ \exists x. 1 \leq x \leq 10 \land y = x^5 \]

The claim is true only if there both is a value in the range, and that value to the fifth power is equal to the program variable \( y \). Using implication here would almost certainly not give the intended meaning:
the assertion would be always true (consider any value for which the left clause is false). To simplify the formulas, it is common to write both of these forms of formulas by using a dot after the restriction on the domain of quantification, where this is interpreted as appropriate, depending on the type of quantifier, as in

$$\forall x \in \{1 \leq x \leq 10, y \neq x^2\}$$

Although a full development of the machinery of predicate logic is not attempted here, a few of the concepts are presented below, to establish a common frame of reference that will be helpful later in describing the properties of specification methods.

The syntax of a logical system is generally defined through rules of grammar, e.g., using a BNF notation. The syntactically legal statements are known as well-formed formulas, abbreviated wff's. The well-formed formulas are assigned a meaning through a mapping function (known as an interpretation) to a value in a semantic domain. The interpretation associates a value from the semantic domain with each formula, usually recursively by assigning meaning to the primitive variables, and then showing the meaning of a more complex formula as a function of the meanings of the subformulas. A propositional symbol $p$, possibly representing a semantic assertion such as $5 > 3$, is associated either with the value true or the value false from the semantic domain, so there are two possible interpretations. There are four possible interpretations for the well-formed formula $p \rightarrow q$, where $p$ and $q$ are propositional variables, each interpretation corresponding to a possible pair of truth values for $p$ and $q$. Three interpretations give the value true, and only the interpretation for which $p$ is true and $q$ is false is mapped to the value false for the implication. Sometimes the interpretations are restricted to those of special interest. Thus if we are interpreting a syntax for Roman numerals, the interpretation in which “V” is mapped to the integer value 5 is really the only one we need consider.

A logical formula is valid if it has the value true in every possible assignment of values to its constituent parts. It is inconsistent if it is false for every possible interpretation. Many well-formed formulas are neither valid nor inconsistent, but rather are true for some interpretations, and false for others.
We also say that a formula $Q$ is a *semantic consequence* of the collection of formulas $P$ (written $P \models Q$) if for all interpretations for which the formulas of $P$ are *true*, the formula $Q$ is also *true*. Note that (as in a *false* left-hand clause of an implication), if the conjunction of the formulas of $P$ is inconsistent, $Q$ can be arbitrary and still be a semantic consequence of $P$.

A *proof system* can be associated with a logic and allows deriving some of the well-formed formulas of the logic. Usually proof systems consist of axioms and deduction rules. The axioms are simply formulas in the logic, and are automatically derived in the proof system. The deduction rules allow deriving additional formulas by stating that if some collection of formulas have been derived as axioms or by applying deduction rules, then a new formula $P$ can be derived. A well-formed formula is said to be *proven* if it can be derived from axioms, or by repeated applications of the deduction rules.

Again, this can be generalized by adding *premises* to the logical proof system, where these are just another collection of well-formed formulas that are temporarily viewed as axioms for the duration of the proof. Then if the premises $P$ are assumed or can themselves be derived the notation $P \vdash Q$ means that $Q$ can be derived using the proof system with the premises $P$.

A proof system is *sound* if only semantically *true* formulas can be derived from it. If we include the generalization to semantic consequences, this can be written as “if $P \vdash Q$, then $P \models Q$.” This is a basic requirement from any proof system. The system is *complete* if every *true* formula can be derived in the proof system (“if $P \models Q$, then $P \vdash Q$”). Unfortunately, most logical systems with semantic domains sufficiently rich to be interesting and expressive must have proof systems that are incomplete if they are also sound. This follows from the revolutionary incompleteness theorem due to Kurt Godel, showing that any proof system including arithmetic must be incomplete if it is to be sound.
2.3 Input/Output assertions

Two first-order logic predicates are sufficient to specify the functional requirements of a program: the input and the output assertions. The former restricts the input values of the variables to those satisfying the predicate, and thus only involves the possible initial values of part of the state. The output assertion is dependent on the input values and the (final) output values, and is true for the acceptable pairs of values. The truth value of the output assertion is irrelevant for those inputs for which the input assertion is false.

To indicate that the values of the variable could change during the computation, the symbolic representation of the input values are usually given a special symbol. One possibility is to use $x_0$ for the initial value of the variable $x$. Another is to use $x$ as the initial value, and $x'$ to denote the final value of the variable $x$. This latter convention has gained in popularity over the years, and it will appear in several specification methods we consider.

In flowchart programs, the two assertions are merely attached to the program, the input specification immediately after the start statement, and the output specification immediately before the halt statement. In 1969, C.A.R. Hoare introduced a more structured approach [15] using a logic of programs that mixes together logical assertions and Pascal-like programs. The notation

$$\{P\}S\{Q\}$$

(known as a Hoare assertion) is commonly used today to denote the assertion that if predicate $P$ is true of the state before computation of the code $S$, then if $S$ terminates, the predicate $Q$ is true of the resulting state. The assertion that $S$ indeed does terminate for every computation is made separately, in words, without any special formalism.

The property expressed by a Hoare assertion $\{P\}S\{Q\}$ is known as partial correctness. It means that whenever a terminating state of the code $S$ is reached for an initial state satisfying $P$, then $Q$ will be true. Such an assertion is a safety property, because it states only what must be true of the states that do occur, but does not guarantee that any particular state will ever be reached. In particular, it does not
guarantee that there really are states that satisfy either $P$ or $Q$, or in which the program terminates.

Termination is not expressed explicitly, and is viewed as a separate, built-in requirement. Termination is therefore the only liveness assertion handled by this approach, and is in fact sufficient because of the limiting assumptions under which this method is used. When more general reactive systems are to be specified, more powerful specification methods must be used.

Already in this simple specification method, both the advantages and potential pitfalls of formal specification can be seen. Consider a program to sort an array of numbers. The specification should only express that the input is an array, say $a$ of size $n$, and the output is a sorted version of the input, say $b$. Other aspects of the program, including which sorting method is used, are irrelevant, at least for this specification. But if our vocabulary does not include a primitive $\text{sorted}(a, b)$ to express that the array $b$ is a sorted version of $a$, then this must be expressed in normal mathematic notation. For example, we might write

$$\forall i. 1 \leq i < n. b[i] \leq b[i + 1]$$

However, this only expresses that the output array must have monotonically increasing values. A program that assigned zero to every array element would satisfy such a specification.

Clearly, the output must be connected to the input: the output array is a permutation of the input array. Again, if we have a primitive $\text{permutation}(a, b)$ we are done, but if not, we might write

$$\forall i. 1 \leq i \leq n. \exists j. 1 \leq j \leq n. b[j] = a[i]$$

This specification seems to give what we want: that every value of the input appears in the output. However, this only captures what we mean if there are no repeated values. Otherwise, the number of appearances of some values could differ in $a$ and in $b$.

The problem of ensuring that the formal notation captures the intuitive intent of the requirements is common to many specification methods, and each attempts to deal with the difficulty in different ways. The need to be precise forces a careful and critical reading of the formal requirement, including the question of how repeated values are to
be treated: should they be forbidden in the input requirement, or is there any special requirement from equally valued elements? (One possibility is to require a stable sorting method in which equally valued elements appear in the same order in the sorted array as in the input.) A specification when repetitions are not allowed would be:

\[
\{ \forall i, j. 1 \leq i \leq n \land 1 \leq j \leq n \land i \neq j. \ a[i] \neq a[j] \}
\]

\[
S
\]

\[
\{ \forall i. 1 \leq i < n. b[i] \leq b[i + 1] \land \forall i. 1 \leq i \leq n. \exists j. 1 \leq j \leq n. b[j] = a[i] \}
\]

This is more difficult to understand than we might like. Specifications in this form also may force us to determine the names of crucial variables at an earlier stage than we might prefer. For example, a program to find whether an element is present in an array \(a\) could have the specification

\[
\{ \text{true} \}
\]

\[
S
\]

\[
\{ \text{found} \rightarrow (\exists j. 1 \leq j \leq n. x = a[j]) \land \neg \text{found} \rightarrow (\forall j. 1 \leq j \leq n. x \neq a[j]) \}
\]

An input specification of \text{true} means that the program will accept any values for the values of the variables before the program is executed. The output specification means that if the value of the variable \text{found} is \text{true} upon termination of the code \(S\), there is an index of the array \(a\) such that \(a[j] = x\), while if \text{found} is \text{false}, \(x\) differs from every value in the array \(a\) for indices between 1 and \(n\).

The dilemma of formal specifications is seen clearly in both of the examples above: on the one hand, it is difficult to write down exactly what we wish to express intuitively. The formalism seems to make irritating demands on the specifier. On the other hand, during the stage of writing the specification important questions are clarified. For example, the issue of what is expected from equal values in sorting is dealt with. Without a specification it is likely that the question would be overlooked until after the program is written. At that stage, if the implemented program does not do what is desired, it may be prohibitively expensive to change.
2.4 Annotated programs and invariants

A specification related to input/output Hoare assertions uses programs annotated with additional assertions. That is, assertions appear in brackets interspersed among the statements of the program. Such annotations have been used as a shorthand form to express the intermediate assertions used in the proof of correctness of a program with respect to its specification given in Hoare assertions. The interpretation of an annotated program is that each assertion in brackets is required to be true whenever the control of the program is at the point in the program where the assertion appears.

An assertion that is true whenever control is at the point where the assertion appears is known as an invariant. The input and output assertions thus become special cases of invariants that appear at the beginning and end of the program, respectively. Every assertion in an annotated program is intended to be an invariant. Note that such a specification is only reasonable if the code of the program already has been written, or at least the form of the implementation has been fixed, since the assertions are defined relative to a location in the code.

In Figure 2.1 an annotated program to compute the factorial of an input \(n\) is given. The annotations show the input specification, the invariant after the initialization part, an invariant that captures the relation among the variables in the loop, and the output assertion.

As another extension, it is also possible to define an invariant independently of a control location in a program. An assertion is called a global invariant of a program if it is true in every state of the program during its execution. If we are able to express the location of the program’s control as a predicate about the state, it is always possible to go from an invariant at a control location to a global invariant by using implication. For example, one treatment of control locations is to assume that there are labels in the program, and the program is in a state where the transition to the next state corresponds to executing a statement labeled \(L\) exactly when the assertion \(control = L\) is true. Then

\[(control = L) \Rightarrow P\]

is a global invariant if and only if \(P\) is an invariant at location \(L\).
{n \geq 1}
\begin{align*}
x &:= 1; \\
y &:= 0; \\
\end{align*}
\begin{align*}
\{n \geq 1 \land x = 1 \land y = 0\}
\end{align*}
while ( y < n )
\begin{align*}
\{x = y! \land y \leq n\}
\end{align*}
do
\begin{align*}
y &:= y+1; \\
x &:= x \ast y; \\
\end{align*}
od
\begin{align*}
\{x = n!\}
\end{align*}

Figure 2.1: An annotated factorial program

An annotated program can be viewed as a specification of the sections of code between the assertions of the annotation. For every such pair of assertions, a partial correctness claim should hold: if the assertion before the code is assumed true, and if the code does terminate then the assertion after the code is required to be true of the state reached. Thus partially implemented systems can be specified by annotating a program skeleton, where a procedure name or just a name serving as a placeholder for code is used instead of actual instructions between assertions. As in partial correctness, only safety properties can be expressed.

2.5 Proving correctness

If an annotated program contains an assertion within every loop of the program, and for every path from an assertion to the next assertion only straight-line code appears, then showing that the partial correctness assertions hold is equivalent to a (generalized) inductive proof that the input-output assertions of the entire program are true. The induction is over the progress of the computation. The inductive hypothesis is
that all states reached up to and including the state at the beginning of the sequence being treated satisfy the associated assertions. Using this assumption, we then show that the states until the end of the sequence satisfy the associated assertions.

If we annotate a program intended to determine whether a value \( x \) appears in an array \( a \) (and if so, find the index), we might have a program of the form

\[
\{ n \geq 1 \} \\
S_0 \\
\{ i = 1 \land \neg\text{found} \}
\]

\( \text{while } B \text{ do} \)

\[
\{ 1 \leq i \land (\text{found} \rightarrow x = a[i] \land i \leq n) \land (\neg\text{found} \rightarrow \forall j. 1 \leq j < i. a[j] \neq x) \}
\]

\( S_1 \)

\( \text{od} \)

\[
\{(\text{found} \rightarrow 1 \leq i \leq n \land x = a[i]) \land (\neg\text{found} \rightarrow \forall j. 1 \leq j \leq n. x \neq a[j]) \}
\]

In this case, the specification of \( S_0 \) is to initialize the index \( i \) and the flag \( \text{found} \), while the annotation of the loop (that appears after the line with \( \text{while} \)) is the precondition of the body of the loop (\( S_1 \)), reached only when the condition \( B \) is also \text{true}. That annotation is also the postcondition of \( S_1 \) and is thus known as the loop invariant. The body of the loop thus has to advance towards the termination condition of the loop by increasing \( i \), and reestablish the loop invariant for the new values of the variables, under the assumption that \( B \) and the invariant were \text{true} before the body was executed. If \( B \) is \text{false} for the values after executing the body of the loop, the loop invariant along with \( \neg B \) should imply the output specification. The code for \( S_0, B, \) and \( S_1 \) are almost dictated by the (very detailed) annotation. For example, it is easy to check that an appropriate \( B \) could be \( i \leq n \land \neg\text{found} \).

The Hoare logic notation is intended to allow proving properties of programs analogously to proofs of properties in logic. A collection of axioms and deduction rules is defined for the ‘mixed’ notation, and the proof system is shown to be semantically sound. That is, any proof of a
Hoare logic assertion done by applying the axioms and deduction rules is valid.

A typical collection of axioms and deduction rules for a simple language of while programs is given in Figure 2.2. The rule at the upper right is known as the post rule, because it allows strengthening the postcondition (substituting a more specific assertion $Q$ that follows from the assertion $R$ proven above the line). The rule below is called the pre rule since it allows weakening the precondition (substituting a more general precondition $P$ in place of an assertion $R$ that it implies). The other rules are named after the control construct that appears in them below the line (the semicolon, if-then-else, and while rules, and the assignment axiom). Note that the deduction rules can be viewed as an indirect definition of the meaning of the language symbols. For example, a semicolon between two statements, that we are used to interpreting as a sequencing instruction, here means only that we are allowed to conclude $\{P\} S_1; S_2\{Q\}$ whenever we are able to show for some assertion $R$ that $\{P\} S_1\{R\}$ and that $\{R\} S_2\{Q\}$. Similarly, if we
are able to show that
\[ \{P \land B\} S_1\{Q\} \]
and that
\[ \{P \land \neg B\} S_2\{Q\} \]
it is reasonable to conclude that
\[ \{P\} \text{if } B \text{ then } S_1 \text{ else } S_2\{Q\}. \]

The rule involving a **while** loop embodies the reasoning seen in the annotated example, but is more difficult to apply than the other rules. This is because the number of times a loop will be repeated depends on the data values when it is executed. In a static proof, an inductive hypothesis—the loop invariant—is used to induct on the progress of the loop. This is the assertion \( P \) in the rule.

Finding an appropriate assertion to allow proving the desired properties is often difficult. Such a \( P \) must have sufficient information to allow proving the implication in the first line of the rule, that if the condition \( B \) for following the loop does not hold then the required postcondition \( Q \) follows from the inductive assertion \((\neg B \land P \Rightarrow Q)\), and \( P \) must also restrict the values in the precondition of the second line, that \( \{P \land B\} S\{P\} \). However, it cannot be so strong that the appearance of \( P \) in the postcondition of the second line is unprovable. This is the nonalgorithmic part of a proof of correctness using this method. For infinite state programs there must be some aspect that is nonalgorithmic, because proving correctness is an undecidable problem: there cannot be one fixed algorithm that would show the correctness or incorrectness of any program-specification pair provided as input to the algorithm.

The assignment axiom can also be seen as an axiomatic definition of the operator: it must ensure that if an assertion is true when the right-hand-side is substituted for all appearances of the variable on the left-hand-side, then after the assignment, the assertion is true for the variable itself. This rule may seem unduly complicated, but the apparently simpler rule \( \{true\} x := e \{x = e\} \) is simply invalid if \( x \) itself can appear within the expression \( e \) (as in \( x := x + 1 \)).

These rules can be used just as in standard logic to produce a proof that a program satisfies a partial order specification. In Figure 2.3 a
formal proof of the partial correctness of the factorial program from Figure 2.1 is given, using the deductive rules and axioms of Figure 2.2. Note that a program with simple variables is proven, since we have not seen the more complex rules needed to treat arrays and indices.

As in proofs of logical assertions, the final form in which the proof is presented is clearly not the way that the proof is developed. In the case of Hoare logic when a program is already available, the first step requires writing the input/output specification as Hoare assertions. Then the key invariants, and especially the loop invariants should be annotated in the program. A full annotation then can be derived by ‘dragging’ the assertions to neighboring annotation points. Finally, a linear representation of the proof can be derived. We shall not pursue this subject further here, but it remains an active area of research. Sophisticated methods exist for extending these basic ideas to more complex program structures, and to provide tools to aid in automation of such proofs.

Because almost every program involves arithmetic and integer or real variables, the limitations of predicate calculus show that we cannot have a complete proof system, simply because some assertion about $P$ or $Q$ cannot be shown either true or false, irregardless of the Hoare
logic. For this reason the notion of relative completeness has been introduced. This means that a collection of Hoare axioms and deduction rules is complete if we assume that any set of values can be expressed as an assertion in the underlying logic and can be proven true in that underlying logic if it is indeed valid. That is, any incompleteness is not "the fault" of the Hoare rules, but can be traced to the incompleteness of the underlying logic.

To show soundness and relative completeness for any collection of proof rules and axioms given in Hoare logic notation, a separate semantic definition for the constructs in the language is needed. This is clearly beyond the scope of this book, since our primary interest is not in the semantics of programming languages, or even proofs. Still, an axiomatic definition can be seen as a specification of a language construct, and suggestions for new language constructs are sometimes given proof rules more as a specification than in the hope that formal proofs will be conducted of programs using the construct.

The collection of rules in Figure 2.2 is sound and relatively complete for while programs of this simple form. When some richer program constructs are included, the Hoare logics are not even relatively complete.

Specifications of programs or systems using input/output assertions are inadequate for reactive or concurrent systems, since intermediate states usually need to be explicitly considered. Moreover, additional liveness properties besides termination must be specified, and these are not treated within the formalism.

Proofs of correctness have also not proven practical in large programs because of the need to annotate the program so precisely. The loop invariants have proven especially difficult to find. However, this notation should be recognized as an effective method for the restricted class of problems and requirements that it can describe. In later chapters we will again briefly relate to how the specification methods can be incorporated into software tools for proving the correctness of implementations. Both deductive techniques, as seen here, and exhaustive checks of all possible states in finite-state implementations are used for this purpose.
2.6 Bibliographic remarks

A few standard texts on mathematical logic, especially as applied in computer science, are [7, 6, 27, 1]. A detailed proof of Gödel’s Incompleteness Theorem can be found in [27].

A precise, mathematical notation for specifications was first used in proving the correctness of programs, beginning in the 1960’s with the work of Robert Floyd and C.A.R. Hoare [8, 15]. The role of invariants and annotated programs in program development were emphasized by David Gries [11] and Edsger Dijkstra [5]. The view of a program construct as a predicate transformer is due to Dijkstra, who presented weakest precondition and strongest postcondition transformers between predicates.

The presentation of annotated programs was also seen in the context of a proof system for shared memory concurrent programs, in the work of Susan Owicki and David Gries [29].
Bibliography


