Five

Advanced Topics in Software Engineering

In this part of *Software Engineering: A Practitioner's Approach,* we consider a number of advanced topics that will extend your understanding of software engineering. In the chapters that follow, we address the following questions:

- What notation and mathematical preliminaries ("formal methods") are required to formally specify software?
- What key technical activities are conducted during the cleanroom software engineering process?
- How is component-based software engineering used to create systems from reusable components?
- What technical activities are required for software reengineering?
- What are the future directions of software engineering?

Once these questions are answered, you'll understand topics that may have a profound impact on software engineering over the next decade.

CHAPTER

28 FORMAL METHODS

KEY CONCEPTS
constructive specification
data invariant
formal specification
logic operators
OCL
operations
pre- and postconditions
schemas
set operators
states
Z language

Software engineering methods can be categorized on a "formality" spectrum that is loosely tied to the degree of mathematical rigor applied during analysis and design. For this reason, the analysis and design methods discussed earlier in this book fall at the informal end of the spectrum. A combination of diagrams, text, tables, and simple notation is used to create analysis and design models, but little mathematical rigor has been applied.

We now consider the other end of the formality spectrum. Here, a specification and design are described using a formal syntax and semantics that specify system function and behavior. The specification is mathematical in form (e.g., predicate calculus can be used as the basis for a formal specification language).

In his introductory discussion of formal methods, Anthony Hall [HAL90] states:

> Formal methods are controversial. Their advocates claim that they can revolutionize [software] development. Their detractors think they are impossibly difficult. Meanwhile, for most people, formal methods are so unfamiliar that it is difficult to judge the competing claims.

In this chapter, we explore formal methods and examine their potential impact on software engineering in the years to come.

QUICK LOOK

What is it? Formal methods allow a software engineer to create a specification that is more complete, consistent, and unambiguous than those produced using conventional methods. Set theory and logic notation are used to create a clear statement of facts (requirements). This mathematical specification can then be analyzed to improve (or even prove) correctness and consistency. Because the specification is created using mathematical notation, it is inherently less ambiguous than informal modes of representation.

Who does it? A specially trained software engineer creates a formal specification.

Why is it important? In safety-critical or mission-critical systems, failure can have a high price. Lives may be lost or severe economic consequences can arise when computer software fails. In such situations, it is essential that errors are uncovered before software is put into operation. Formal methods reduce specification errors dramatically and, as a consequence, serve as the basis for software that has very few errors once the customer begins using it.

What are the steps? The notation and heuristics of sets and constructive specification—set operators, logic operators, and sequences—form the basis of formal methods. Formal methods define the data invariant, states, and operations for a system function by translating informal requirements for the problem into a more formal representation.

What is the work product? A specification represented in a formal language such as OCL or Z is produced when formal methods are applied.

How do I ensure that I've done it right? Because formal methods use discrete mathematics as the specification mechanism, logic proofs can be applied to each system function to demonstrate that the specification is correct. However, even if logic proofs are not used, the structure and discipline of a formal specification will lead to improved software quality.

28.1 Basic Concepts

The Encyclopedia of Software Engineering [MAR94] defines formal methods in the following manner:

A method is formal if it has a sound mathematical basis, typically given by a formal specification language. This basis provides a means of precisely defining notions like consistency and completeness, and more relevantly, specification, implementation and correctness.

The desired properties of a formal specification—consistency, completeness, and lack of ambiguity—are the objectives of all specification methods. However, the use of formal methods results in a much higher likelihood of achieving these ideals. The formal syntax of a specification language (Section 28.4) enables requirements and design to be interpreted in only one way, eliminating ambiguity that often occurs when a natural language (e.g., English) or a graphical notation must be interpreted by a reader. The descriptive facilities of set theory and logic notation (Section 28.2) enable clear statement of facts (requirements). To be consistent, facts stated in one place in a specification should not be contradicted in another place. Consistency is ensured by mathematically proving that initial facts can be formally mapped (using inference rules) into later statements within the specification.

"Formal methods have tremendous potential for improving the clarity and precision of requirements specifications, and in finding important and subtle errors."

Steve Easterbrook et al.

Completeness is difficult to achieve, even when formal methods are used. Some aspects of a system may be left undefined as the specification is being created; other characteristics may be purposely omitted to allow designers some freedom in choosing an implementation approach; and finally, it is impossible to consider every operational scenario in a large, complex system. Things may simply be omitted by mistake.

Although the formalism provided by mathematics has an appeal to some software engineers, others (some would say, the majority) look askance at a mathematical view of software development. To understand why a formal approach has merit, we must first consider the deficiencies associated with less formal approaches.

28.1.1 Deficiencies of Less Formal Approaches[1]

The methods discussed for analysis and design in Parts 2 and 3 of this book make heavy use of natural language and a variety of graphical notations. Although careful application of analysis and design methods coupled with thorough review can and does lead to high-quality software, sloppiness in the application of these methods can create a variety of problems. A system specification can contain contradictions, ambiguities, vagueness, incomplete statements, and mixed levels of abstraction.

ADVICE

Although a good document index cannot eliminate contradictions, it can help to uncover them. Consider creating an index for specifications and other documents.

Contradictions are sets of statements that are at variance with each other. For example, one part of a system specification may state that the system must monitor all the temperatures in a chemical reactor while another part, perhaps written by another person may state that only temperatures occurring within a certain range are to be monitored.

Ambiguities are statements that can be interpreted in a number of ways. For example, the following statement is ambiguous:

> The operator identity consists of the operator name and password; the password consists of six digits. It should be displayed on the security VDU and deposited in the login file when an operator logs into the system.

In this extract, does the word *it* refer to the password or the operator identity?

Vagueness often occurs because a system specification is a very bulky document. Achieving a high level of precision consistently is an almost impossible task.

> **"Making mistakes is human. Repeating 'em is too."**
>
> **Malcolm Forbes**

Incompleteness is one of the most frequently occurring problems with system specifications. For example, consider the functional requirement:

> The system should maintain the hourly level of the reservoir from depth sensors situated in the reservoir. These values should be stored for the past six months.

ADVICE

Effective formal technical reviews can eliminate many of these problems. However, some will not be uncovered. Be on the lookout for deficiencies during design, code, and test.

This describes the main data storage part of a system. If one of the commands for the system was

> The function of the AVERAGE command is to display on a PC the average water level for a particular sensor between two times.

and assuming that no more detail was presented for this command, the details of the command would be seriously incomplete. For example, the description of the command does not include what should happen if a user of a system specifies a time that was more than six months before the current hour.

1 This section and others in the first part of this chapter have been adapted from work contributed by Darrel Ince for the European edition of the fifth edition of *Software Engineering: A Practitioner's Approach.*

Mixed levels of abstraction occur when very abstract statements are intermixed randomly with statements that are at a much lower level of detail. While both types of statements are important in a system specification, specifiers often manage to intermix them to such an extent that it becomes very difficult to see the overall functional architecture of a system.

28.1.2 Mathematics in Software Development

Mathematics has many useful properties for the developers of large systems. One is that it can succinctly and exactly describe a physical situation, an object, or the outcome of an action. A specification of a computer-based system can be developed using specialized mathematics in much the same way that an electrical engineer can use mathematics to describe a circuit.[2]

Mathematics supports abstraction and thus is an excellent medium for modeling. Because it is an exact medium there is little possibility of ambiguity. Specifications can be mathematically validated for contradictions and incompleteness, and vagueness can be eliminated. In addition, mathematics can be used to represent levels of abstraction in a system specification in an organized way.

Finally, mathematics provides a high level of validation when it is used as a software development medium. It is possible to use a mathematical proof to demonstrate that a design matches a specification and that program code is a correct reflection of a design.

28.1.3 Formal Methods Concepts

The aim of this section is to present the main concepts involved in the mathematical specification of software systems, without encumbering the reader with too much mathematical detail. To accomplish this, we use a few simple examples.

Example 1: a symbol table. A program is used to maintain a symbol table. Such a table is used frequently in many different types of applications. It consists of a collection of items without any duplication. An example of a typical symbol table is shown in Figure 28.1. It represents the table used by an operating system to hold the names of the users of the system. Other examples of tables include the collection of names of staff in a payroll system or the collection of names of computers in a network communications system.

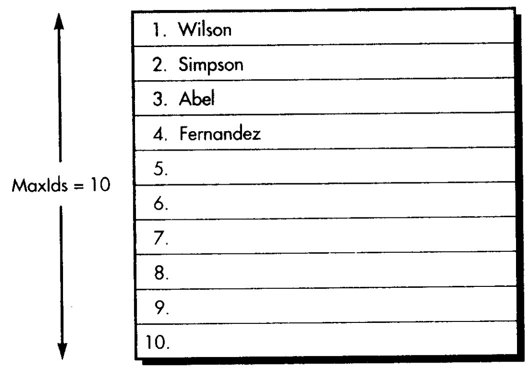
KEY POINT
A data invariant is a set of conditions that are true throughout the execution of the system that contains a collection of data.

Assume that the table presented in this example consists of no more than *MaxIds* members of staff. This statement, which places a constraint on the table, is a component of a condition known as a *data invariant*—an important idea that we shall return to throughout this chapter.

2 A word of caution is appropriate at this point. The mathematical system specifications that are presented in this chapter are not as succinct as a mathematical specification for a simple circuit. Software systems are notoriously complex, and it would be unrealistic to expect that they could be specified in one line of mathematics.

FIGURE 28.1

A symbol table



ADVICE

Another way of looking at the notion of state is to say that data determines state. That is, you can examine data to see what state the system is in.

A data invariant is a condition that is true throughout the execution of the system that contains a collection of data. The data invariant that holds for the symbol table just discussed has two components: (1) that the table will contain no more than *MaxIds* names and (2) that there will be no duplicate names in the table. In the case of the symbol table program, this means that no matter when the symbol table is examined during execution of the system, it will always contain no more than *MaxIds* staff identifiers and will contain no duplicates.

Another important concept is that of a *state*. Many formal languages, such as OCL (Section 28.5) , use the notion of a state as it was discussed in Chapters 7 and 8; that is, a system can be in one of several states, each representing an externally observable mode of behavior. However, a different definition for the term *state* is used in the Z language (Section 28.6). In Z (and related languages), the state of a system is represented by the system's stored data (hence, Z suggests a much larger number of states, representing each possible configuration of the data). Using the latter definition in the example of the symbol table program, the state is the symbol table.

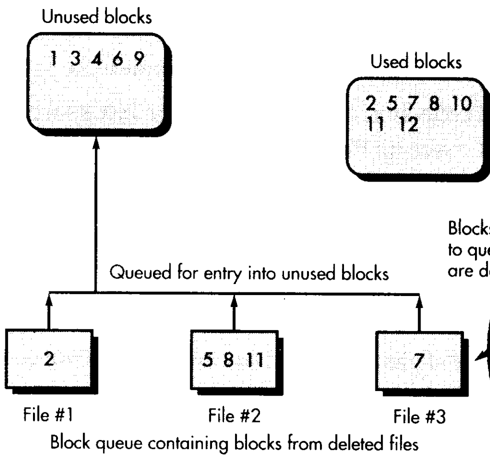
The final concept is that of an *operation*. This is an action that takes place within a system and reads or writes data. If the symbol table program is concerned with adding and removing staff names from the symbol table, then it will be associated with two operations: an operation to *add* a specified name to the symbol table and an operation to *remove* an existing name from the table.[3] If the program provides the facility to check whether a specific name is contained in the table, then there would be an operation that would return some indication of whether the name is in the table.

Three types of conditions can be associated with operations: invariants, preconditions, and postconditions. An *invariant* defines what is guaranteed not to change. For example, the symbol table has an invariant that states that the number of elements is always less than or equal to *MaxIds*. A *precondition* defines the circum-

3 It should be noted that adding a name cannot occur in the *full* state and deleting a name is impossible in the *empty* state.

FIGURE 28.2

A block handler


Unused blocks
1 3 4 6 9
Used blocks
2 5 7 8 10 11 12
Blocks are released to queue when files are deleted
Queued for entry into unused blocks
2
5 8 11
7
File #1
File #2
File #3
Block queue containing blocks from deleted files

stances in which a particular operation is valid. For example, the precondition for an operation that adds a name to the staff identifier symbol table is valid only if the name that is to be added is not contained in the table and also if there are fewer than *MaxIds* staff identifiers in the table. The *postcondition* of an operation defines what is guaranteed to be true upon completion of an operation. This is defined by its effect on the data. In the example of an operation that adds an identifier to the staff identifier symbol table, the postcondition would specify mathematically that the table has been augmented with the new identifier.

ADVICE

Brainstorming techniques can work well when you must develop a data invariant for a reasonably complex function. Have each member of the software team write down bounds, restrictions, and limitations for the function and then combine and edit.

Example 2: a block handler. One of the more important parts of a computer's operating system is the subsystem that maintains files created by users. Part of the filing subsystem is the *block handler.* Files in the file store are composed of blocks of storage that are held on a file storage device. During the operation of the computer, files will be created and deleted, requiring the acquisition and release of blocks of storage. To cope with this, the filing subsystem will maintain a reservoir of unused (free) blocks and keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks. This is shown in Figure 28.2. In this figure, a number of components are shown: the reservoir of unused blocks, the blocks that currently make up the files administered by the operating system, and those blocks that are waiting to be added to the reservoir. The waiting blocks are held in a queue, with each element of the queue containing a set of blocks from a deleted file.

For this subsystem the state is the collection of free blocks, the collection of used blocks, and the queue of returned blocks. The data invariant, expressed in natural language, is:

- No block will be marked as both unused and used.

- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.
- No elements of the queue will contain the same block numbers.
- The collection of used and unused blocks will be the total collection of blocks that make up files.
- The collection of unused blocks will have no duplicate block numbers.
- The collection of used blocks will have no duplicate block numbers.

Some of the operations associated with the data invariant are: *add()* a collection of blocks to the end of the queue, *remove()* a collection of used blocks from the front of the queue and place them in the collection of unused blocks, and *check()* whether the queue of blocks is empty.

The precondition of the first operation is that the blocks to be added must be in the collection of used blocks. The postcondition is that the collection of blocks is now found at the end of the queue. The precondition of the second operation is that the queue must have at least one item in it. The postcondition is that the blocks must be added to the collection of unused blocks. The final operation—checking whether the queue of returned blocks is empty—has no precondition. This means that the operation is always defined, regardless of what value the state is. The postcondition delivers the value *true* if the queue is empty and *false* otherwise.

In the examples noted in this section, we introduce the key concepts of formal specification. But we do so without emphasizing the mathematics that are required to make the specification formal. In Section 28.2, we consider these mathematics.

28.2 Mathematical Preliminaries

To apply formal methods effectively, a software engineer must have a working knowledge of the mathematical notation associated with sets and sequences and the logical notation used in predicate calculus. The intent of the section is to provide a brief introduction. For a more detailed discussion the reader is urged to examine books dedicated to these subjects (e.g., [WIL87], [GRI93], and [ROS95]).

28.2.1 Sets and Constructive Specification

A *set* is a collection of objects or elements and is used as a cornerstone of formal methods. The elements contained within a set are unique (i.e., no duplicates are allowed). Sets with a small number of elements are written within curly brackets (braces) with the elements separated by commas. For example, the set

{C++, Smalltalk, Ada, COBOL, Java}

contains the names of five programming languages.

The order in which the elements appear within a set is immaterial. The number of items in a set is known as its *cardinality.* The # operator returns a set's cardinality. For example, the expression

$$\#\{A, B, C, D\} = 4$$

implies that the cardinality operator has been applied to the set shown with a result indicating the number of items in the set.

What is constructive set specification?

There are two ways of defining a set. A set may be defined by enumerating its elements (this is the way in which the sets just noted have been defined). The second approach is to create a *constructive set specification*. The general form of the members of a set is specified using a Boolean expression. Constructive set specification is preferable to enumeration because it enables a succinct definition of large sets. It also explicitly defines the rule that was used in constructing the set. Consider the following constructive specification example:

$$\{n : \mathbb{N} \mid n < 3 \bullet n\}$$

This specification has three components, a signature, $n : \mathbb{N}$, a predicate $n < 3$, and a term, n. The *signature* specifies the range of values that will be considered when forming the set; the *predicate* (a Boolean expression) defines how the set is to be constricted; and, finally, the *term* gives the general form of the item of the set. In the example above, $\mathbb{N}$ stands for the natural numbers; therefore, natural numbers are to be considered. The predicate indicates that only natural numbers less than 3 are to be included; and the term specifies that each element of the set will be of the form n. Therefore, this specification defines the set

$$\{0, 1, 2\}$$

When the form of the elements of a set is obvious, the term can be omitted. For example, the preceding set could be specified as

$$\{n : \mathbb{N} \mid n < 3\}$$

ADVICE

Knowledge of set operations is indispensable when formal specifications are developed. Spend the time to familiarize yourself with each, if you intend to apply formal methods.

All the sets that have been described here have elements that are single items. Sets can also be made from elements that are pairs, triples, and so on. For example, the set specification

$$\{x, y : \mathbb{N} \mid x + y = 10 \bullet (x, y^2)\}$$

describes the set of pairs of natural numbers that have the form (x, y^2) and where the sum of x and y is 10. This is the set

$$\{(1, 81), (2, 64), (3, 49), \ldots\}$$

Obviously, a constructive set specification required to represent some component of computer software can be considerably more complex than those noted here. However, the basic form and structure remain the same.

28.2.2 Set Operators

A specialized symbology is used to represent set and logic operations. These symbols must be understood by the software engineer who intends to apply formal methods.

The $\in$ operator is used to indicate membership of a set. For example, the expression

$x \in X$

has the value *true* if x is a member of the set X and the value *false* otherwise. For example, the predicate

$12 \in \{6, 1, 12, 22\}$

has the value *true* since 12 is a member of the set.

The opposite of the $\in$ operator is the $\notin$ operator. The expression

$x \notin X$

has the value *true* if x is not a member of the set X and *false* otherwise. For example, the predicate

$13 \notin \{13, 1, 124, 22\}$

has the value *false.*

The operators $\subset$, and $\subseteq$, take sets as their operands. The predicate

$A \subset B$

has the value *true* if the members of the set A are contained in the set B and has the value *false* otherwise. Thus, the predicate

$\{1, 2\} \subset \{4, 3, 1, 2\}$

has the value *true.* However, the predicate

{HD1, LP4, RC5} $\subset$ {HD1, RC2, HD3, LP1, LP4, LP6}

has a value of *false* because the element RC5 is not contained in the set to the right of the operator.

The operator $\subseteq$ is similar to $\subset$. However, if its operands are equal, it has the value *true.* Thus, the value of the predicate

{HD1, LP4, RC5} $\subseteq$ {HD1, RC2, HD3, LP1, LP4, LP6}

is *false,* and the predicate

{HD1, LP4, RC5} $\subseteq$ {HD1, LP4, RC5}

is *true.*

"Mathematical structures are among the most beautiful discoveries made by the human mind."
Douglas Hofstadter

A special set is the empty set ∅. This corresponds to zero in normal mathematics. The *empty set* has the property that it is a subset of every other set. Two useful identities involving the empty set are

$\varnothing \cup A = A$ and $\varnothing \cap A = \varnothing$

for any set *A*, where ∪ is known as the *union operator*, sometimes known as *cup*; ∩ is the *intersection operator*, sometimes known as *cap*.

The union operator takes two sets and forms a set that contains all the elements in the set with duplicates eliminated. Thus, the result of the expression

{Filel, File2, Tax, Compiler} ∪ {NewTax, D2, D3, File2}

is the set

{Filel, File2, Tax, Compiler, NewTax, D2, D3}

The intersection operator takes two sets and forms a set consisting of the common elements in each set. Thus, the expression

{12, 4, 99, 1} ∩ {1, 13, 12, 77}

results in the set {12, 1}.

The set *difference operator*, \, as the name suggests, forms a set by removing the elements of its second operand from the elements of its first operand. Thus, the value of the expression

{New, Old, TaxFile, SysParam} \ {Old, SysParam}

results in the set {New, TaxFile}.

The value of the expression

{a, b, c, d} ∩ {x, y}

will be the empty set ∅. The operator always delivers a set; however, in this case there are no common elements between its operands, so the resulting set will have no elements.

The final operator is the *cross product*, ×, sometimes known as the *Cartesian product*. This has two operands which are sets of pairs. The result is a set of pairs where each pair consists of an element taken from the first operand combined with an element from the second operand. An example of an expression involving the cross product is

{1, 2} × {4, 5, 6}

The result of this expression is

{(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6)}

Notice that every element of the first operand is combined with every element of the second operand.

A concept that is important for formal methods is that of a *powerset*. A powerset of a set is the collection of subsets of that set. The symbol used for the powerset operator in this chapter is $\mathbb{P}$. It is a unary operator that, when applied to a set, returns the set of subsets of its operand. For example,

$$\mathbb{P}\,\{1, 2, 3\} = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

since all the sets are subsets of {1, 2, 3}.

28.2.3 Logic Operators

Another important component of a formal method is *logic:* the algebra of true and false expressions. The meaning of common logical operators is well understood by every software engineer. However, the logic operators that are associated with common programming languages are written using readily available keyboard symbols. The equivalent mathematical operators to these are

$\wedge$	and
$\vee$	or
$\neg$	not
$\Rightarrow$	implies

Universal quantification is a way of making a statement about the elements of a set that is true for every member of the set. Universal quantification uses the symbol, $\forall$. An example of its use is

$$\forall\, i, j \colon \mathbb{N} \bullet i > j \Rightarrow i^2 > j^2$$

which states that for every pair of values in the set of natural numbers, if i is greater than j, then i^2 is greater than j^2.

28.2.4 Sequences

A sequence is a mathematical structure that models the fact that its elements are ordered. A sequence s is a set of pairs whose elements range from 1 to the highest-number element. For example,

{(1, Jones), (2, Wilson), (3, Shapiro), (4, Estavez)}

is a sequence. The items that form the first elements of the pairs are collectively known as the *domain* of the sequence, and the collection of second elements is known as the *range* of the sequence. In this book, sequences are designated using angle brackets. For example, the preceding sequence would normally be written as ⟨Jones, Wilson, Shapiro, Estavez⟩.

Unlike sets, duplication in a sequence is allowed, and the ordering of a sequence is important. Therefore,

$\langle$Jones, Wilson, Shapiro$\rangle \neq \langle$Jones, Shapiro, Wilson$\rangle$

The empty sequence is represented as $\langle\ \rangle$.

A number of sequence operators are used in formal specifications. Catenation, $\frown$, is a binary operator that forms a sequence constructed by adding its second operand to the end of its first operand. For example,

$\langle 2, 3, 34, 1\rangle \frown \langle 12, 33, 34, 200\rangle$.

results in the sequence $\langle 2, 3, 34, 1, 12, 33, 34, 200\rangle$.

Other operators that can be applied to sequences are *head, tail, front,* and *last.* The operator *head* extracts the first element of a sequence; *tail* returns with the last $n - 1$ elements in a sequence of length n; *last* extracts the final element in a sequence; and *front* returns with the first $n - 1$ elements in a sequence of length n. For example,

$head\,\langle 2, 3, 34, 1, 99, 101\rangle = 2$
$tail\,\langle 2, 3, 34, 1, 99, 101\rangle = \langle 3, 34, 1, 99, 101\rangle$
$last\,\langle 2, 3, 34, 1, 99, 101\rangle = 101$
$front\,\langle 2, 3, 34, 1, 99, 101\rangle = \langle 2, 3, 34, 1, 99\rangle$

Since a sequence is a set of pairs, all set operators described in Section 28.2.2 are applicable. When a sequence is used in a state, it should be designated as such by using the keyword *seq*. For example,

$FileList : seq$ FILES
$NoUsers : \mathbb{N}$

describes a state with two components: a sequence of files and a natural number.

28.3 Applying Mathematical Notation for Formal Specification

To illustrate the use of mathematical notation in the formal specification of a software component, we revisit the block handler example presented in Section 28.1.3. To review, an important component of a computer's operating system maintains files that have been created by users. The block handler maintains a reservoir of unused blocks and will also keep track of blocks that are currently in use. When blocks are released from a deleted file they are normally added to a queue of blocks waiting to be added to the reservoir of unused blocks. This has been depicted schematically in Figure 28.2.[4]

A set named *BLOCKS* will consist of every block number. *AllBlocks* is a set of blocks that lie between 1 and *MaxBlocks.* The state will be modeled by two sets and a sequence. The two sets are *used* and *free.* Both contain blocks—the *used* set

4 If your recollection of the block handler example is hazy, please return to Section 28.1.3 to review the data invariant, operations, preconditions and postconditions associated with the block handler.

? How can I represent states and data invariants using the set and logic operators that have already been introduced?

contains the blocks that are currently used in files, and the *free* set contains blocks that are available for new files. The sequence will contain sets of blocks that are ready to be released from files that have been deleted. The state can be described as

$$\begin{array}{l} \textit{used, free}: \mathbb{P}\ \textit{BLOCKS} \\ \textit{BlockQueue}: \textit{seq}\ \mathbb{P}\ \textit{BLOCKS} \end{array}$$

This is very much like the declaration of program variables. It states that *used* and *free* will be sets of blocks and that *BlockQueue* will be a sequence, each element of which will be a set of blocks. The data invariant can be written as

$$\begin{array}{l} \textit{used} \cap \textit{free} = \varnothing \land \\ \textit{used} \cup \textit{free} = \textit{AllBlocks} \land \\ \forall\, i\text{: dom } \textit{BlockQueue} \bullet \textit{BlockQueue}\ i \subseteq \textit{used} \land \\ \forall\, i, j\text{: dom } \textit{BlockQueue} \bullet i \,\Box\, j \Rightarrow \textit{BlockQueue}\ i \cap \textit{BlockQueue}\ j = \varnothing \end{array}$$

WebRef
Extensive information on formal methods can be found at **www.afm.sbu.ac.uk.**

The mathematical components of the data invariant match four of the bulleted, natural-language components described earlier. The first line of the data invariant states that there will be no common blocks in the used collection and free collections of blocks. The second line states that the collection of used blocks and free blocks will always be equal to the whole collection of blocks in the system. The third line indicates the *i*th element in the block queue will always be a subset of the used blocks. The final line states that, for any two elements of the block queue that are not the same, there will be no common blocks in these two elements. The final two natural language components of the data invariant are implemented by virtue of the fact that *used* and *free* are sets and therefore will not contain duplicates.

The first operation we shall define is one that removes an element from the head of the block queue. The precondition is that there must be at least one item in the queue:

$$\#\text{BlockQueue} > 0,$$

The postcondition is that the head of the queue must be removed and placed in the collection of free blocks and the queue adjusted to show the removal:

$$\begin{array}{l} \textit{used}' = \textit{used} \setminus \textit{head BlockQueue} \land \\ \textit{free}' = \textit{free} \cup \textit{head BlockQueue} \land \\ \textit{BlockQueue}' = \textit{tail BlockQueue} \end{array}$$

A convention used in many formal methods is that the value of a variable after an operation is primed. Hence, the first component of the preceding expression states that the new used blocks (*used'*) will be equal to the old used blocks minus the blocks that have been removed. The second component states that the new free blocks (*free'*) will be the old free blocks with the head of the block queue added to it. The third component states that the new block queue will be equal to the tail of the old value of the block queue; that is, all elements in the queue apart from the first one.

A second operation adds a collection of blocks, *Ablocks,* to the block queue. The precondition is that *Ablocks* is currently a set of used blocks:

? How do I represent pre- and post-conditions?

$$Ablocks \subseteq used$$

The postcondition is that the set of blocks is added to the end of the block queue, and the set of used and free blocks remains unchanged:

$$BlockQueue' = BlockQueue \frown \langle \text{Ablocks} \rangle \land$$
$$used' = used \land$$
$$free' = free$$

There is no question that the mathematical specification of the block queue is considerably more rigorous than a natural language narrative or a graphical model. The additional rigor requires effort, but the benefits gained from improved consistency and completeness can be justified for many types of applications.

28.4 Formal Specification Languages

A formal specification language is usually composed of three primary components: (1) a *syntax* that defines the specific notation with which the specification is represented, (2) *semantics* to help define a "universe of objects" [WIN90] that will be used to describe the system, and (3) a *set of relations* that define rules that indicate which objects properly satisfy the specification.

The syntactic domain of a formal specification language is often based on a syntax that is derived from standard set theory notation and predicate calculus. For example, variables such as *x, y,* and *z* describe a set of objects that relate to a problem (sometimes called the *domain of discourse*) and are used in conjunction with the operators described in Section 28.2. Although the syntax is usually symbolic, icons (e.g., graphical symbols such as boxes, arrows, and circles) can also be used, if they are unambiguous.

The *semantic domain* of a specification language indicates how the language represents system requirements. For example, a programming language has a set of formal semantics that enables the software developer to specify algorithms that transform input to output. A formal grammar (such as BNF) can be used to describe the syntax of the programming language. However, a programming language does not make a good specification language because it can represent only computable functions. A specification language must have a semantic domain that is broader; that is, the semantic domain of a specification language must be capable of expressing ideas such as, "For all *x* in an infinite set *A,* there exists a *y* in an infinite set *B* such that the property *P* holds for *x* and *y*" [WIN90]. Other specification languages apply semantics that enable the specification of system behavior. For example, a syntax and semantics can be developed to specify states and state transition, and events, along with their effect on state transition, synchronization and timing.

28.5.2 An Example Using OCL

In this section, OCL is used to help formalize the specification of the block handler example, introduced in Section 28.1.3. The first step is to develop a UML model. For this example we start with the class diagram found in Figure 28.3. This diagram specifies many relationships among the objects involved; however we must add OCL expressions to ensure that implementers of the system know more precisely what they must ensure remains true as the system runs.

The OCL expressions we will add correspond to the six parts of the invariant discussed in Section 28.1.3. For each, we will repeat the invariant in English and then give the corresponding OCL expression. It is considered good practice to provide English text along with the formal logic; doing so helps the reader to understand the logic, and also helps reviewers to uncover mistakes, e.g., situations where the English and the logic do not correspond.

1. No block will be marked as both unused and used.

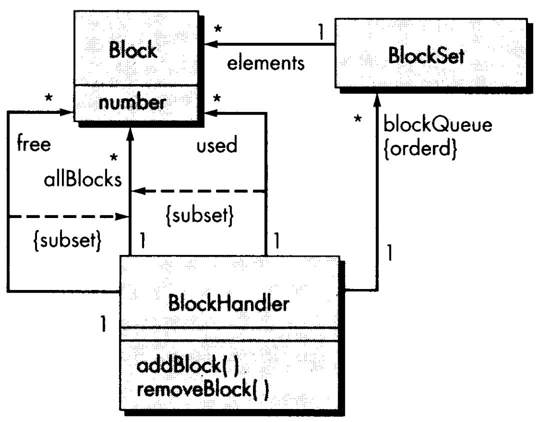
   ```
   context BlockHandler inv:
       (self.used->intersection(self.free)) ->isEmpty()
   ```

   Note that each expression starts with the keyword **context**. This indicates the element of the UML diagram that the expression constrains. Alternatively, the software engineer could place the constraint directly on the UML diagram, surrounded by braces {}. The keyword **self** here refers to the instance of **BlockHandler**; in the following, as is permissible in OCL, we will omit the **self**.

2. All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.

   ```
   context BlockHandler inv:
       blockQueue->forAll(aBlockSet | used->includesAll(aBlockSet ))
   ```

FIGURE 28.3



Class diagram for a block handler

3. No elements of the queue will contain the same block numbers.

```
context BlockHandler inv:
    blockQueue->forAll(blockSet1, blockSet2 |
        blockSet1 <> blockSet2 implies
        blockSet1.elements.number->excludesAll(blockSet2.elements.number))
```

   The expression before **implies** is needed to ensure we ignore pairs where both elements are the same block.

4. The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.

```
context BlockHandler inv:
    allBlocks = used->union(free)
```

5. The collection of unused blocks will have no duplicate block numbers.

```
context BlockHandler inv:
    free->isUnique(aBlock | aBlock.number)
```

6. The collection of used blocks will have no duplicate block numbers.

```
context BlockHandler inv:
    used->isUnique(aBlock | aBlock.number)
```

OCL can also be used to specify preconditions and postconditions of operations. For example, consider operations that remove and add sets of blocks to the queue. Note that the notation **x@pre** indicates the object **x** as it existed *prior* to the operation; this is opposite to mathematical notation discussed earlier, where it is the **x** *after* the operation that is specially designated (as **x'**).

```
context BlockHandler::removeBlocks()
    pre: blockQueue->size() >0
    post: used = used@pre - blockQueue@pre->first() and
        free = free@pre->union(blockQueue@pre->first()) and
        blockQueue = blockQueue@pre->excluding(blockQueue@pre->first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
    pre: used->includesAll(aBlockSet.elements)
    post: (blockQueue.elements = blockQueue.elements@pre
        ->append(aBlockSet))and
        used = used@pre and
        free = free@pre
```

OCL is a modeling language, but it has all of the attributes of a formal language. OCL allows the expression of various constraints, pre- and postconditions, guards, and other characteristics that relate to the objects represented in various UML models.

28.6 The Z Specification Language

Z (properly pronounced as "zed") is a specification language that has evolved over the past two decades to become widely used within the formal methods community. The Z language applies typed sets, relations, and functions within the context of first-order predicate logic to build *schemas*—a means for structuring a formal specification.

WebRef
Detailed information about the Z language can be found at **www-users.cs.york.ac.uk/~susan/abs/z.htm.**

28.6.1 A Brief Overview of Z Syntax and Semantics

Z specifications are organized as a set of schemas—a boxlike structure that introduces variables and specifies the relationship between these variables. A *schema* is essentially the formal specification analog of the programming language component. In the same way that components are used to structure a system, schemas are used to structure a formal specification.

A schema describes the stored data that a system accesses and alters. In the context of Z, this is called the "state." This usage of the term *state* in Z is slightly different from the use of the word in the rest of this book.[6] In addition, the schema identifies the operations that are applied to change state and the relationships that occur within the system. The generic structure of a schema takes the form:

```
—— schemaName ————————————
   declarations
——————————————————————————
   invariant
——————————————————————————
```

where declarations identify the variables that comprise the system state and the invariant imposes constraints on the manner in which the state can evolve. A summary of Z language notation is presented in Table 28.2.

28.6.2 An Example Using Z

In this section, we use the Z specification language to model the block handler example, introduced earlier in this chapter. The following example of a schema describes the state of the block handler and the data invariant:

—— BlockHandler ————————————

$used, free : \mathbb{P}\ BLOCKS$

$BlockQueue : \mathrm{seq}\ \mathbb{P}\ BLOCKS$

$used \cap free = \varnothing\ \wedge$

6 Recall that in other chapters *state* has been used to identify an externally observable mode of behavior for a system.

TABLE 28.2 SUMMARY OF Z NOTATION

Z notation is based on typed set theory and first-order logic. Z provides a construct, called a *schema*, to describe a specification's state space and operations. A schema groups variable declarations with a list of predicates that constrain the possible value of a variable. In Z, the schema X is defined by the form

```
———X——————————————————————————————
    declarations
——————————————————————————————————
    predicates
——————————————————————————————————
```

Global functions and constants are defined by the form

```
    declarations
——————————————————————————————————
    predicates
```

The declaration gives the type of the function or constant, while the predicate gives it value. Only an abbreviated set of Z symbols is presented in this table.

Sets:

$S : \mathbb{P}\, X$	*S* is declared as a set of *X*s.
$x \in S$	*x* is a member of *S*.
$x \notin S$	*x* is not a member of *S*.
$S \subseteq T$	*S* is a subset of *T*: Every member of *S* is also in *T*.
$S \cup T$	The union of *S* and *T*: It contains every member of *S* or *T* or both.
$S \cap T$	The intersection of *S* and *T*: It contains every member of both *S* and *T*.
$S \setminus T$	The difference of *S* and *T*: It contains every member of *S* except those also in *T*.
$\varnothing$	Empty set: It contains no members.
$\{x\}$	Singleton set: It contains just *x*.
$\mathbb{N}$	The set of natural numbers 0, 1, 2, ...
$S : \mathbb{F}\, X$	*S* is declared as a finite set of *X*s.
$\max(S)$	The maximum of the nonempty set of numbers *S*.

Functions:

$f{:}X ⤔ Y$	*f* is declared as a partial injection from *X* to *Y*
$\mathrm{dom}\ f$	The domain of *f*: the set of values *x* for which *f*(*x*) is defined.
$\mathrm{ran}\ f$	The range of *f*: the set of values taken by *f*(*x*) as *x* varies over the domain of *f*.
$f \oplus \{x \mapsto y\}$	A function that agrees with *f* except that *x* is mapped to *y*.
$\{x\} ⩤ f$	A function like *f*, except that *x* is removed from its domain.

Logic:

$P \wedge Q$	*P* and *Q*: It is true if both *P* and *Q* are true.
$P \Rightarrow Q$	*P* implies *Q*: It is true if either *Q* is true or *P* is false.
$\theta\, S' = \theta\, S$	No components of schema *S* change in an operation.

used $\cup$ *free* = *AllBlocks* $\wedge$

$\forall i$: **dom** *BlockQueue* • *BlockQueue i* $\subseteq$ *used* $\wedge$

$\forall i, j$: dom *BlockQueue* • *i* ▯ *j* => *BlockQueue i* $\cap$ *BlockQueue j* = $\varnothing$

———————————————————————————

As we have noted, the schema consists of two parts. The part above the central line represents the variables of the state, while the part below the central line describes the data invariant. Whenever the schema specifies operations that change the state, it is preceded by the Δ symbol. The following example of a schema describes the operation that removes an element from the block queue:

——— RemoveBlocks ———————————

Δ *BlockHandler*

———————————————————

$\#\text{BlockQueue} > 0,$

$used' = used \setminus head\ BlockQueue \wedge$

$free' = free \cup head\ BlockQueue \wedge$

$BlockQueue' = tail\ BlockQueue$

———————————————————

The inclusion of Δ *BlockHandler* results in all variables that make up the state being available for the *RemoveBlocks* schema and ensures that the data invariant will hold before and after the operation has been executed.

The second operation, which adds a collection of blocks to the end of the queue, is represented as

———AddBlocks———————————

Δ BlockHandler

Ablocks? : BLOCKS

———————————————————

$Ablocks? \subseteq used$

$BlockQueue' = BlockQueue \quad \langle\text{Ablocks?}\rangle \wedge$

$used' = used \wedge$

$free' = free$

———————————————————

By convention in Z, an input variable that is read but does not form part of the state is terminated by a question mark. Thus, Ablocks?, which acts as an input parameter, is terminated by a question mark.

SOFTWARE TOOLS

Formal Methods

Objective: The objective of formal methods tools is to assist a software team in specification and correctness verification.

Mechanics: Tools mechanics vary. In general, tools assist in specification and automated correctness proving, usually by defining a specialized language for theorem proving. Many tools are not commercialized and have been developed for research purposes.