Teaching Mathematics to Software Engineers

Jeannette M. Wing[*]

School of Computer Science, Carnegie Mellon University,
Pittsburgh, PA 15213, USA

Abstract. Based on my experience in teaching formal methods to practicing and aspiring software engineers, I present some of the common stumbling blocks faced when writing formal specifications. The most conspicuous problem is learning to abstract. I address all these problems indirectly by giving a list of hints to specifiers. Thus this paper should be of interest not only to teachers of formal methods but also to their students.

1 Context: Teach *What* to *Whom*?

Let me explain this paper's title, and in particular what I mean by "mathematics" and "software engineers." By "mathematics" I mean the mathematical foundations and techniques that underlie a wide range of *formal methods*. The foundations are mathematical logic and discrete mathematics. The techniques are specification and reasoning. Specification is a synthesis process involving the construction of a mathematical model or theory from an informally described set of concepts (e.g., an English-language description of a system's requirements). In this paper, when I say "specify" ("specification") I mean "*formally* specify" ("*formal* specification"). Reasoning is an analysis process involving proving properties about a system from a formal specification; the proofs themselves may be informal or formal, but their logical basis is formal. This paper focuses more on specification than reasoning.
By "software engineer" I mean to include practicing software engineers in industry, who may be furthering their education by taking courses while on the job, or taking a leave from work to get an advanced (typically, Master's) degree; graduate students who design and build large software systems, typically as an integral component of their Ph.D. thesis work;

[*] This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

and undergraduate students who want to learn some basic software engineering principles before working at their first job.

My target teaching audience is not students of mathematics but students who are or will be software engineers. What and how to teach mathematical foundations to them differs from the contents and methods taught in traditional courses on logic and discrete mathematics. Indeed this observation led a group of us (Alan Brown, David Garlan, Daniel Jackson, James Tomayko, and me) to redesign the core curriculum of the Carnegie Mellon Master's of Software Engineering Program to include a course called "Models of Software Systems" [7, 6]. This course aims exactly at teaching the mathematical foundations and techniques relevant to the practice of software engineering.

This paper is based on the following experiences, starting with the most influential:

- Co-teaching (with David Garlan) the CMU MSE "Models" course during the Fall 1993 and Fall 1994 semesters. A total of thirty-six Master's and advanced undergraduate students enrolled. Z [19] and CSP [8] are the predominate formal methods used.
- Working one-on-one with CMU Ph.D students, mine and those of other faculty members, in the "Programming Systems" area. I have worked closely with or served as specification consultant for over twenty students in the past ten years.
- Working one-on-one with a Japanese industrial visitor, a software engineer in control systems, this past year. He is using the Larch/C Interface Language (LCL).
- Co-teaching (with Daniel Jackson) the CMU MSE "Analysis of Software Artifacts" course during the Spring 1995 semester. Students use the formalisms introduced in "Models" and analyze specifications of pieces of real systems. Fifteen students are enrolled.
- Teaching a Ph.D.-level graduate course at CMU, "Reasoning about Concurrent and Distributed Systems," Spring 1994, based on the MIT course, "Principles of Computer Systems," developed by Butler Lampson and William Weihl [12]. Students use the specification language *Spec* (roughly a combination of Modula-3 and Dijkstra's guarded command language) to describe system interfaces. About ten students either were enrolled or audited during the course of the term.
- Co-teaching (with John Guttag) a graduate course at MIT, "Specifications in Software Development" during the Fall 1992 semester. We covered Larch [9] in some detail. Ten Master's and Ph.D. students enrolled.

- Working as a laboratory assistant (while I was a graduate student) for the industrial short course, "Design and Implementation of Modular Software." Barbara Liskov and John Guttag were the instructors. We worked together on four incarnations of this course during 1980-1982. We briefly covered an algebraic specification technique.
- Teaching (Spring 1993) and co-teaching (with Bernd Bruegge, Spring 1991) the CMU undergraduate software engineering course. The major part of this course is an integrated class project whose subcomponents are developed by separate teams. Students are expected to write stylized, informal interface specifications.

Over the years I have been accumulating specification hints that I give to students in response to common problems and recurrent questions. So, I will use an indirect way to explain to teachers of formal methods the lessons I have learned; in the following four sections I speak directly to students of formal methods and exemplify the kinds of stumbling blocks they encounter when first using formal methods. Herein I use "you" to refer to the reader who is the software engineer attempting to write a formal specification.

I've broadly categorized the issues along the following dimensions:

- Figuring out *why* you are going through this specification effort (Section 2). What do you hope to get out of using formalism?
- Figuring out *what* of the system you want *to specify* (Section 3).
- Figuring out *how* to specify (Section 4). The most important hurdle to overcome is learning to abstract. I also give specific suggestions on how to make incremental progress when writing a specification.
- Figuring out *what to write down* (Section 5). Learn a formal method's set of conventions but do not feel constrained by them. Also, we all make logical errors sometimes; I point out some common troublespots in getting the details of a specification right.

I will illustrate my points with examples, usually in Z or Larch. Many actually make more than one point.

I close with a discussion of challenges to the technical community in Section 6.

2 Why Specify?

You should first ask yourself this question, "Why specify?" You might choose to specify because you want additional documentation of your system's interfaces, you want a more abstract description of your system design, or you want to perform some formal analysis of your system. What you write should be determined by what it is you want to do with your specification.

You should then ask yourself "Why *formally* specify?" Your answer determines what is to be formalized, what formal method to use, and what benefits you expect from a formal specification not attainable from just an informal one. When I have asked this question of a software engineer, here are the kinds of responses I have heard:

- Showing that a property holds globally of the entire system.
  - I want to characterize the "correctness condition" I can promise the user of my system.
  - I want to show this property is really a system invariant.
  - I want to show my system meets some high-level design criteria.
- Error handling
  - I want to specify what happens if an error occurs.
  - I want to specify the right thing happens if an error occurs.
  - I want to make sure this error never occurs.
- "Completeness"
  - I want to make sure that I've covered all the cases, including error cases, for this protocol.
  - I'd like to know that this language I've designed is computationally complete.
- Specifying interfaces.
  - I'd like to specify a hierarchy of C++ classes.
  - I'd like a more formal description of this system's user interface.
- Getting a handle on complexity.
  - The design is getting too complicated. I can't fit it all in my head. I need a way to think about it in smaller pieces.
- Change control.
  - Everytime I change one piece of code I need to know what other pieces are affected. I'd like to know where else to look without looking at all modules, without looking at all the source code.

Judicious use of mathematics, e.g., by applying formal methods, can help address all these problems to varying levels of detail and rigor.

3 What to Specify?

Formal methods are not to the point where an entire large, software system can be formally specified. You may be able to specify one aspect of it, e.g., its functionality or its real-time behavior; you may be able to specify many aspects of a part of it, e.g., specifying both functionality and real-time behavior of its safety-critical part. In practice, you may care to specify only one aspect of a part of a system anyway.

In writing a specification, you should know whether the specification is describing *required* or *permitted* behavior. Must or may? Since a specification can be viewed as an abstraction of many possible, legitimate implementations, you might most naturally associate a specification with describing permitted behavior. An implementation may have any of the behaviors permitted by the specification, but the implementor is not required to realize all. For example, a nondeterministic *choose* operation specified for sets will have a deterministic implementation. However, the expression "software system requirements" suggests that a customer may in fact *require* certain behavior. For example, in specifying an abstract data type's interface, the assumption is that all, not some subset, of the operations listed must be implemented.

Once it is clear what you hope to gain from the specification process, you can turn to determining exactly what should be formalized.

In increasing order of level of detail, you might want to formalize a global *correctness condition* for the system, one or more system *invariants*, the *observable behavior* of a system, or *properties* about entities in a system.

Correctness Conditions

You usually have some informal notion of a global *correctness condition* that you expect your system to uphold. It might be something as standard as serializability, cache coherence, or deadlock freedom. Or, it might be very specific to the protocol or system at hand. If it is standard, then very likely someone else has developed a formal model for characterizing a system and a logic such that the correctness condition can be formally stated and proven. E.g., serializability has been thoroughly studied by the database community from all angles, theoretical to practical. If your correctness condition can be cast in terms of a well-known theory, it pays to reuse that work and not invent from scratch.

If it is not standard then an informal statement of the correctness condition should drive the formalization of the system model and expression of the correctness condition. For example, in work by Mummert et al. [16], the authors started with this informal statement of cache coherence for a distributed file system:

> If a client believes that a cached file is valid then the server that is the authority on that file had better believe that the file is valid.

They developed a system model (a state machine model) and logic (based on the logic of authentication [4]) that enabled them to turn the informal statement into the following formal statement:

For all clients C, servers S, and objects d for which S is the repository, if C **believes** $\mathit{valid}(d_C)$ then S **believes** $\mathit{valid}(d_C)$

where clients, servers, objects, repository, **believes** and *valid* are formally defined concepts. The point is that the formal statement does not read too differently from the informal one.

Keep in mind this rule-of-thumb when formalizing from an informal statement: Let the things you want to describe formally drive the description of the formal model. There is a tendency to let the formal method drive the description of the formal model; you end up specifying what you can easily specify using that method. That is fine as far as it goes, but if there are things you cannot say or that are awkward to express using that method, you should not feel bound by the method. Invent your own syntax (to be defined later), add auxiliary definitions, or search for a complementary method.

The process of constructing a formal model of a system and formally characterizing the intended correctness condition can lead to surprises. More than once in my work with Ph.D. students who were formalizing the systems they were building, they had to back off from their expected and desired correctness condition by realizing that it was too strong, not always guaranteed (e.g., not guaranteed for some failure case or for a "fast-path" case), or only locally true (holds for a system component but not the entire system). Correctness conditions for distributed systems are likely to be weaker than expected or desired because of the presence of failures (nodes or links crashing) and transmission delays; it is likely that there are "windows of vulnerability" during which the correctness condition cannot be guaranteed.

Invariants

The most common way to characterize certain kinds of correctness conditions is as a *state invariant*. An invariant is a property that does not change as the system goes from state to state. Remember also:

- An invariant is just a predicate. If you define some appropriate assertion language, it is usually not a big deal to express an invariant formally.
- "True" is an invariant of any system. It's the weakest invariant and hence not a very satisfying one; you probably want to say something more interesting about your system. If "true" ends up being your strongest invariant, revisit your system design.
- An invariant can serve multiple purposes. It is usually used to pare down a state space to the states of interest. For example, it can be

used to characterize the set of *reachable* states or the set of *acceptable/legal* ("good") states. (These two sets are not always the same. For example, you might want the set of acceptable states to be a subset of the set of reachable ones.) *Representation invariants* are used to define the domain of an abstraction function, used when showing that one system "implements" another [13].

- Different formal methods treat invariants differently. (See *Implicit versus Explicit* in Section 5.1 for an elaboration of this point.) Make sure you understand invariants in the context of the formal method you are using.
- Hard questioning of system invariants can lead to radically new designs.

To illustrate the last point, consider this example from the garbage collection community. One class of copying garbage collection algorithms relies on dividing the heap into two semi-spaces, *to-space* and *from-space*; in one phase of these algorithms, objects are copied from from-space to to-space [2]. Traditional copying garbage collection algorithms obey a "to-space invariant": The user accesses objects only in to-space. Nettles and O'Toole observed that breaking this invariant and maintaining an alternative "from-space invariant" (the user accesses objects only in from-space) leads to simpler designs that are much easier to implement, analyze, and measure [17]. This observation led to a brand new class of garbage collection algorithms.

Observable Behavior

Given that state invariants are a good way to characterize desired system properties, formalizing the state transitions will allow you to prove that the invariants are maintained. When you specify state transitions, what you are specifying is the behavior of the system as it interacts with its environment, i.e., the system's observable behavior.

It might seem obvious that what you want to specify is the observable behavior of a system, but sometimes when you are buried in the details of the task of specifying, you forget the bigger picture. Suppose you take a state machine approach to modeling your system. Here is a general approach to specifying observable behavior:

1. Identify the level of abstraction (see Section 4.1) at which you are specifying the system. This level determines the interface boundary that you are specifying; it determines what is or is not observable. For example, a bus error at the hardware level is not expected to be an observable event in the execution of an text formatter like Word, but `core dumped` is certainly an observable event when using a text editor like **emacs**.

2. Characterize the observable entities in a state at that given level of abstraction. These are your state variables or objects. This step forces you to identify the relevant abstract *types* of your system (See the section on *Properties of State Entities* below.)
3. Characterize a set of initial states, and if appropriate, a set of final states.
4. Identify the operations that can access or modify the observable entities. These define your state transitions.
5. For each operation, characterize its observable effect on the observable state entities. For example, use Z schemas, Larch interfaces, or VDM pre/post-conditions.

Observable behavior should include any change in state that is observable to the user. If you are specifying an operation, then the kinds of observable state changes include changes in value to state objects, observable changes in the store (new objects that appear and old objects that disappear), objects returned by the operation, and signaled exceptions or errors.

Another way to think about observable behavior is to think about *observable equivalence* [15, 11, 10]. Ask "Can I distinguish between these two things?" where "things" might be states, individual entities in a state, traces of a process, or behavior sets of a process, depending on what you are specifying. If the answer is "yes," then there must be way to tell them apart (perhaps by using unique names or perhaps by defining an *equal* operation); if "no," then there must not be any way for the observer to tell them apart.

Properties of State Entities

The most important property to state of any entity in a system is its *type*. This statement is true regardless of the fine distinctions between the different type systems that different formal methods and specification (and programming) languages have. Since for a specification we are not concerned about compile-time and/or run-time costs of checking types, there is never a cost incurred in documenting the type of an entity in a specification.

Since a type can be viewed as an abbreviation for a little theory, declaring an entity's type is a succinct way of associating a possibly infinite set of properties with the entity in one or two words. A truly powerful abstraction device!

For entities that are "structured" objects (e.g., an object that is a collection of other objects), when determining its type, the kinds of distinguishing properties include:

- Ordering. Are elements ordered or unordered? If ordering matters, is the order partial or total? Are elements removed FIFO, LIFO, or by priority?

- Duplicates. Are duplicates allowed?
- Boundedness. Is the object bounded in size or unbounded? Can the bound change or it is fixed at creation time?
- Associative access. Are elements retrieved by an index or key? Is the type of the index built-in (e.g., as for sequences and arrays) or user-definable (e.g., as for symbol tables, hash tables)?
- Shape. Is the structure of the object linear, hierarchical, acyclic, n-dimensional, or arbitrarily complex (e.g., graphs, forests)?

For entities that are relations, the kinds of distinguishing properties include whether the relation is a function (many-to-one), partial, finite, defined for only a finite domain, surjective, injective, bijective, and/or any (meaningful) combination of these.
Finally, algebraic properties help characterize any relational entity or any function or relation defined on a structured entity. The common algebraic properties include: idempotency, reflexivity, symmetry, transitivity, commutativity, associativity, distributivity, existence of an identity element, and existence of an inverse relation or function. Algebras are a well-known mathematical model for both abstract data types and processes [8, 15, 1]. For example, this algebraic equation characterizes the idempotency of inserting the same element into a set multiple times:

$$insert(insert(s, e), e) = insert(s, e)$$

and this characterizes *insert*'s commutativity property:

$$insert(insert(s, e_1), e_2) = insert(insert(s, e_2), e_1)$$

And, since processes can be viewed as denoting structured objects (e.g., sets of traces), it makes sense to ask about whether algebraic properties hold for operations on processes. For example, for CSP processes, parallel composition is both commutative and associative:

$$P \parallel Q = Q \parallel P$$
$$P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$$

4 How to Specify?

Given that you understand why you are specifying and what it is you want to specify, in what ways should you try to think about the system so that you can begin to specify and then make progress in writing your specification? Not surprisingly, we rely on the tried and true techniques: *abstraction* and *decomposition*. In specifying large, complex systems, abstraction is useful for focusing your attention to one level of detail at a time; decomposition, for one small piece of the system (at a given abstraction level) at a time. Both enable local reasoning.

4.1 Learn to Abstract: Try Not to Think Like a Programmer

The skill that people find the most difficult to acquire is the ability to abstract. One aspect of learning to abstract is being able to think at a level higher than programmers are used to.

Try to think definitionally not operationally.
A student said the following to me when trying to explain his system design:

> If you do this and then that and then this and then that, you end up in a good state. But if you do this and then that and then this, you end up in a bad state.

Rather than thinking of what characterizes all good states, people find it easier to think about whether a particular sequence of operations leads to a good or bad state. This operational approach means ending up trying to enumerate all possible interleavings; this enumeration process quickly gets out of control, which is typically when a student will come knocking at my door for help. This problem is related to understanding invariants (see *Invariants* in Section 3). Invariably, the very first thing I need to teach students when I work with them one-on-one is what an invariant is.

Try not to think computationally.[2]
When writing specifications abstraction is intellectually liberating because you are not bound to think in terms of computers and their computations.
The following predicate

$$s = s' \frown \langle e \rangle$$

might appear in the post-condition of the specification of a *remove* operation on sequences. Here, s stands for the sequence's initial value; s', its final value; e, the element removed and returned. You most naturally might read "=" as assignment (especially if you are a C programmer) and not as a predicate symbol used here to relate values of objects in two different states; you may need to stare at these kinds of predicates for a while before realizing the assertional nature (and power) of logic.

Try constructing theories, not just models.
Building models is an abstraction process; but defining a theory takes a different kind of abstraction skill. When you construct a model of a system in terms of mathematical structures like sets, sequences, and relations, you get all properties of sets and relations "for free." This has

[2] Another way of saying the same thing as above.

the advantage that you do not have to spell them out every time you specify a system, but the disadvantage that some of those properties are irrelevant to your system. Thus, in a model-based constructive approach, you also need to provide a way to say which properties about the standard mathematical structures are irrelevant. (You might do this stripping away of properties in terms of invariants.) For example, you might specify a stack in terms of a sequence, where the top of the stack corresponds to one end of the sequence. Then, you need not only to state which end of the sequence serves as the top of the stack, but also to eliminate some sequence properties, e.g., being able to index into a sequence or concatenate two sequences, because they have no relevance for stacks. Alternatively, in a theory-based approach you state explicitly exactly what properties you want your system to have. Any model that satisfies that theory is deemed to be acceptable. For example, the essence of stacks is captured by the well-known equations:

$$pop(push(s,e)) = s$$
$$top(push(s,e)) = e$$

Sequences, or any other particular mathematical structure, do not enter the picture at all.

Like many, you may find methods like Z and VDM appealing because they encourage a model-based rather than theory-based approach to specification. You can build up good intuition about your system if you have a model in hand. However, to practice learning how to abstract, try writing algebraic or axiomatic assertions about the model.

4.2 How To Proceed: Incrementally

At any given level of abstraction, we ignore some detail about the system below. You might feel anxious to specify everything for fear of being "incomplete." Learning to abstract means learning when it is okay to leave something unspecified. This aspect of the abstraction process also allows incremental specification. In any case, it is probably better to specify something partially than not at all.

Here are four common and important examples of incremental abstraction techniques: (1) first assume something is true of the input argument and capture this assumption in a pre-condition, then weaken the pre-condition; (2) first handle the normal case, then the failure case; (3) first ignore the fact that ordering (or no duplicates, etc.) matters, then strengthen the post-condition; (4) first assume the operation is atomic, then break it into smaller atomic steps. Let's look in turn at each of these examples in their generality and in more detail.

Use pre-conditions.
Putting your "programmers" cap on, think of pre-conditions in the context of procedure call. A pre-condition serves two purposes: an obligation on the caller to establish before calling the procedure and an assumption the implementor can make when coding the procedure.
More generally, pre-conditions are a way of specifying assumptions about the environment of a system component. Thus, such assumptions can and should be spelled out and written down explicitly, and in so doing, you can specify and reason about a piece of the system without having to think about the entire system all at once. Thus pre-conditions enable partial specification, incremental design, and local reasoning—all attractive means of dealing with the complexity of large software systems.
One technical difficulty that trips some people is what it means or what happens if a pre-condition is not met. In many specification techniques (like Z and Larch), when an operation's pre-condition is not met, the interpretation is "all bets are off." The interpretation is that the pre-condition is a *disclaimer.*[3] In other words, the operation is free to do anything, including not terminate, if the pre-condition does not hold. The technical justification is that when an operation is specified using pre- and post-conditions, the logical interpretation of the specification is an implication:

$$pre \Rightarrow post$$

When the pre-condition is "false" then the implication is vacuously true, so any behavior should be allowed.
However, a stronger interpretation (e.g., taken by InaJo [18] and I/O automata [14]) is that a pre-condition should be interpreted as a *guard.* No state transition should occur if the pre-condition/guard is not met. Here the interpretation of a pre/post-condition specification is conjunction:

$$pre \wedge post$$

The difference is that under the disclaimer interpretation, for any state s in which the pre-condition does not hold, the state pair, $\langle s, s' \rangle$, for any state s', would be in the state transition relation; under the guard interpretation, no such state pair would be in the relation.[4]
There are other possible interpretations: For example, if the pre-condition is not met, it could mean that the state transition always goes to a special "error" state and termination is guaranteed, or it could mean the state transition leads to either an "error" state or non-termination. The

[3] Thanks to Daniel Jackson for this term.

[4] There is further confusion in interpreting pre-conditions in Z because even though you might write explicitly in your schema the conjunction, $xpre \wedge post$, where $xpre$ is the "explicit" pre-condition, the meaning is the implication, $pre \Rightarrow post$, where pre is the calculated pre-condition and usually not identical to $xpre$ [5].

point is that for the formal method you are using you should understand what it means both when a pre-condition is met and not met.

Finally, in the presence of concurrency, an operation's pre-condition is usually interpreted as a guard. More subtly, the predicate is evaluated in the state in which the operation begins executing, not in the state in which the operation is called. Because of concurrency, a scheduler may delay the start of the execution of an operation to some time after the call of the operation; since there is time between the state in which the operation is called and the state in which it starts executing, an intervening operation (executed by some other process) may change the system's state. Thus, a predicate that holds in the state when the operation is called may no longer hold in the state when the operation begins to execute. The point is to realize that in the presence of concurrency, there may be a change in interpretation of pre-conditions. [5]

Specify errors/exceptions/failures.

It is as important to specify erroneous or exceptional behavior as it is to specify normal behavior. If an operation can lead to an undesired state, you should specify the conditions under which this state is reachable. If you are lucky the specification language has some notational convenience (e.g., Larch's **signals** clause) or prescribed technique (e.g., Z's schema calculus) to remind you to describe error conditions; otherwise, handling errors may have to be disguised in terms of input or output arguments that serve as error flags.

There is a close correlation between pre-conditions and handling errors. Z draws this connection by advocating this convention using schema disjunction:

$$TotalOp = NormalOp \vee ErrorOp$$

where NormalOp is the specification (schema) of the Op operation under normal conditions, and ErrorOp is the specification of Op under the condition in which the pre-condition (which must be *calculated* [19] from NormalOp) does not hold. Thus, TotalOp gives the specification of Op under all possible conditions.

Larch, on the other hand, draws the connection by advocating weakening the pre-condition, e.g., defining it to be equivalent to "true," and correspondingly strengthening the post-condition. Thus,

```
Op = op()
        requires P
        ensures  Q
```

[5] Larch calls the guard a *when-condition* to distinguish it from the standard pre-condition written in a **requires** clause.

turns into:

```
Op = op() signals (error)
        requires true
        ensures if $P$ then $Q$ else signal error
```

For interfaces to distributed systems, you cannot ignore the possibility of failure due to network partitions or crashed nodes. You could abstract from the different kinds of failures by introducing a generic "failure" exception that stands for errors arising from the distributed nature of your system.

The two main points to remember are (1) in support of incremental specification, specify the normal case and then handle the error cases, but (2) do not forget to handle the error cases!

Use nondeterminism.

Introducing nondeterminism is an effective abstraction technique. Nondeterminism permits design freedom and avoids implementation bias.

Nondeterminism may show up in many ways. First, it may be inherent to the behavior of an operation or object. Consider the *choose* operation on sets:

```
choose = op ($s$: set) returns ($e$: elem)
      requires $s \neq \emptyset$
      ensures $e \in s$
```

The post-condition says that the element returned is a member of the set argument; it does not specify exactly which element is returned.

Nondeterminism may result from explicit use of disjunction in a post-condition:

```
light = op() returns ($c$: color)
      ensures $c = red \vee c = amber \vee c = green$
```

or more generally, from the explicit use of an existential quantifier:

```
positively_random = op () returns ($i$: int)
      ensures $\exists\, i \,.\, i > 0$
```

From a state machine model viewpoint (for instance when discussing deterministic and nondeterministic finite state automata), nondeterminism should not be confused with choice. Suppose δ is a state transition relation,

$$\delta : State, Action \rightarrow 2^{State}$$

Then an example of choice is:

$$\delta(s, a_1) = \{t\}$$
$$\delta(s, a_2) = \{u\}$$

which says from state s you can either do the action a_1 (and go to the next state, t), or do the action a_2 (and go to the next state, u). However, an example of nondeterminism is:

$$\delta(s, a_1) = \{t, u\}$$

which says from state s you can do action a_1 and go to either state t or u.

Some formal methods for concurrent systems introduce their own notions of nondeterminism/choice; for example, CSP has two operators, one for internal choice ($\sqcap$) made by the machine and the other for external choice ($\square$) made by the environment[6]. CCS has yet a different way to model nondeterminism.

The two main points are that (1) nondeterminism is a useful and important way to abstract, but (2) be careful to understand any given method's way of modeling nondeterminism and/or choice to use it properly.

Use Atomic Operations

For any system it is important to identify what the *atomic* operations are. At any level of abstraction an atomic operation may be implemented in terms of sequences of lower-level atomic operations (e.g., a *write* operation to a file on disk might be implemented in terms of a sequence of *write* operations to individual disk blocks). Even assignment can be broken down into sequences of loads and stores to/from memory and registers.

For a sequential program, usually it is assumed that each procedure is executed atomically; this assumption is rarely stated explicitly.

For a concurrent system, it is critical to state explicitly what operations are assumed to be atomic. The atomicity of an operation, Op, guarantees that no other operation can interfere with the execution of Op and that you can abstract away from any intermediate (lower-level) state that the operation might actually pass through.

5 What to Write?

With your pen poised over a blank sheet of paper or fingers over your keyboard, you now face the problem of what to write down. If you are using a specific formal method like Z, VDM, or Larch, you must know the syntax and semantics of its specification language. It is not enough to know what the syntactic features are; you need to understand what each means.

Also, embedded within any formal method is an *assertion language*, usually based on some variation of first-order predicate logic. With assertions you nail down precisely your system's behavior. It is in your assertions where the smallest change in syntax can have a dramatic change in semantics. Getting the details of your assertions right is typically when

[6] Hoare calls the former "nondeterministic or" and the latter "general choice."

you discover most of the conceptual misunderstandings of your system's design.

It is important to understand the difference between syntax and semantics. For example, the typical assertion language for algebraic specification languages gives grammatical rules for formulating syntactically legal terms out of function and variable symbols. Each syntactically legal term denotes a value in some underlying algebraic model. For example, the term, $insert(insert(\emptyset, e_1), e_2)$ is a syntactic entity that denotes the set value, $\{e_1, e_2\}$, which is a semantic entity. And so for a standard model of sets, the syntactically different term $insert(insert(\emptyset, e_2), e_1)$ denotes the same semantic set value.

5.1 General Rules-of-Thumb

What typically distinguishes a formal method from mathematics is its methodological aspects, e.g., stylistic conventions. A specification written in the style of a given formal method is usually not just an unstructured set of formulae. Syntactic features make it easier to read the specification (e.g,. the lines in a Z schema), remind the specifier what to write (e.g., the **modifies** clause in Larch), and aid in structuring a large specification into smaller, more modular pieces (e.g., Z schemas, Larch traits).

Implicit vs. Explicit

Most formal methods have well-defined specification languages so the choice of what you explicitly write down is guided by the grammar and constructs of the language.

However, there is a danger of forgetting the power of the unsaid. What is *not* explicitly stated in a specification often has a meaning. A naive specifier is likely to be unaware of these implicit consequences, thereby be in danger of writing nonsense. Here are three examples.

The first example is the *frame* issue. If you are specifying the behavior of one piece of the system in one specification module, you should say what effects that piece has *on the rest* of the system. In some formal methods (e.g., InaJo), you are forced to say explicitly what other pieces of the system do not change (NC''):

$NC''(x_1, \ldots, x_n)$

This is sometimes impractical if n is large, or worse, if you do not or cannot know what the $x_1, \ldots, x_n$ are in advance.

In some methods (e.g., Larch), you say only what may (but is not required to) change; anything not listed explicitly is required *not* to change:

modifies $y_1, \ldots, y_m$

This says $y_1 \ldots y_m$ may change but the rest of the system stays the same.

A subtler point about the Larch **modifies** clause is that there is significance to the omission of the clause. The absence of a **modifies** clause says that no objects may change. Thus, you might write a post-condition that asserts some change in value to an input argument or global; this assertion would be inconsistent with an omitted **modifies** clause.

Z's Δ and Ξ operators on schemas are similar to InaJo's *NC* construct; they allow you to make statements local to individual operations about whether they change certain state variables or not. Use of these schema operators on say the schemas, S_i, leaves implicit the invariant properties of the system captured in S_i. These properties can be made explicit by "unrolling" the schemas S_i.

This feature of Z is related to my second example of implicit vs. explicit specification, which has to do with invariants. In some formal methods like Z, state invariants are stated explicitly. They are a critical part of the specification, i.e., "property" component of a Z schema, and used to help calculate operation pre-conditions. In others like Larch, they are implicit and must be proven, usually using some kind of inductive proof rule. Finally, in others like VDM, they are redundant. They are stated explicitly and contribute to the checklist of proof obligations generated for each operation.

Finally, the third example has to do with implicit quantification. In many algebraic specification languages the i equations in this list

$$e_1$$
$$\ldots$$
$$e_i$$

are implicitly conjoined and quantified as follows:

$$\exists f_1 \ldots \exists f_n \,.\, \forall x_1 \ldots \forall x_m \,.\, e_1 \wedge \ldots \wedge e_i$$

where $f_1 \ldots f_n$ are the function symbols and $x_1 \ldots x_m$ are the variables that appear in $e_1 \ldots e_i$.

This kind of implicit quantification has subtle consequences. Consider the following (incorrect) equational specification of an operation that determines whether one set is a subset of another:

$$s_1 \subseteq s_2 = (e \in s_1 \Rightarrow e \in s_2)$$

What you really mean is:

$$s_1 \subseteq s_2 = \forall e.(e \in s_1 \Rightarrow e \in s_2)$$

but in most algebraic specification languages, writing a quantifier in the equation is syntactically illegal; the tipoff to the error is the occurrence of the free variable e on the righthand side of the first equation.

Auxiliary Definitions

Do not be afraid to use auxiliary definitions:

- To shorten individual specification statements. For example, when argument lists to functions get too long (say, greater than four), then it probably means the function being defined is "doing too much."
- To "chunkify" and enable reuse of concepts. When a long expression (say, involving more than two logical operators and three function symbols) appears multiple (say, more than two) times, then it probably means that chunk of information can be given a name and the name reused accordingly.
- To postpone specifying certain details. When you find yourself going into too much depth while specifying one component of the system in neglect of specifying the rest of the system, then introduce a placeholder term to be defined later.

Notation

You should not feel overly constrained by the specification notation. If there is a concept you want to express and you cannot express it easily in the given notation, invent some convenient syntax, say what you want, and defer giving it a formal meaning till later. Don't get stuck just because your notation is restrictive. On the other hand, don't forget to define your inventions. It may be at odds with the rest of the semantics. If you're lucky, however, you will have thought of a new specification language construct that is more generally useful than for just your problem at hand.

Proofs

Most likely you will not be proving theorems about your system from your specifications, but if you are, the first difficult aspect about doing proofs is knowing how formal to be. For realistic systems and/or large examples, it's impractical to do a completely formal proof, in the strictest sense of "formal" as used in mathematical logic. What you should strive for when writing out an informal proof is to justify each proof step where in principle the step could be formalized.

Given that you are doing only informal proofs, the second difficult aspect is knowing when you can skip steps. Some steps are "obvious" but others are not. Also, what may seem "obvious" often reflects a hole in your argument.

It is possible to do large formal proofs using machine aids like proof checkers and theorem provers. There is of course a tradeoff between the effort needed to learn to use one of these tools and its input language and underlying logic and the benefit gained by doing the more formal proof. If what you are trying to prove is critical, it may pay to invest the time and energy; moreover, this cost need be done only once, the first time. If you plan to do more than one (critical) proof, it may be worth your while. Finally, using machine aids keeps you honest because they do not let you skip steps.

Choosing the degree of formality and how much proof detail to give takes experience and practice, gained by both reading other people's proofs and constructing your own. A background in mathematics usually helps.

There are common proof techniques that you should have in your arsenal: proof by induction, case analysis, proof by contradiction, and equational reasoning (substituting equals for equals). You should be familiar with natural deduction though you probably would use it for only small, local proofs.

Finally, the familiar commuting diagram from mathematics plays a central role in proofs of correctness for software systems. For example, an interpretation for Fig. 1 in the context of state machines is to suppose that f is an action of a concrete machine on the concrete state x. If $\langle x, f, x' \rangle$ is a state transition of the concrete machine, then there exists an abstract action g such that $\langle A(x), g, A(x') \rangle$ is a state transition of the abstract machine.

$$\begin{tikzcd}
y \arrow[r, bend left, "g"] & y' \\
x \arrow[u, "A"] \arrow[r, bend right, "f"'] & x' \arrow[u, "A"']
\end{tikzcd}$$

Fig. 1. A Commuting Diagram

In the context of abstract data types, the interpretation is that given that x is a concrete representation for y, the concrete function f *implements* the abstract function g under the abstraction function A. That is,

$$A(f(x)) = g(A(x))$$

More elaborate diagrams, for example, that allow sequences of actions rather than single actions generalize this basic idea. The "CLInc Stack" case study [3] of proving the correctness of the implementation of a small programming language down to the hardware level relies fundamentally on a stack of commuting diagrams.

5.2 The Details

I now turn to the nitty gritty of specification: getting the technical details right.

Logical Errors
Common logical errors that I have seen specifiers (including myself) make involve implication and quantification.
Implication. Remember that *false* implies anything so that

$$\mathit{false} \Rightarrow \ldots$$

is vacuously true, and that anything implies *true* so that

$$\ldots \Rightarrow \mathit{true}$$

reduces to true.
Quantifiers. Problem spots include nested quantifiers, ordering of quantifiers (especially modal operators for a temporal logic), and combining quantifiers and implication (e.g., what happens to a formula when bringing a quantifier outside an implication). Another confusion arises when qualifying a quantified variable with set membership, $\in$. That is,

$$\forall x \in T \,.\, P(x)$$

translates to

$$\forall x \,.\, x \in T \Rightarrow P(x)$$

but

$$\exists x \in T \,.\, P(x)$$

translates to

$$\exists x \,.\, x \in T \wedge P(x)$$

If you have a complicated predicate with a lot of embedded quantifiers, you may find it helpful to break the predicate into pieces, where each piece is in prenex form and has only one or two quantifiers.

Properties about sets, functions, and relations
When specifying objects such as sets, bags, and sequences that are collections of objects you may be prone to making the following common errors.
Saying

$$x \in s'$$

in the post-condition of an *insert* operation on sets may be too weak. It does not say that elements in the set that were originally in s are still there.
Saying

$$s' = s - \{x\}$$

in the post-condition of a *remove* operation on sets is too weak. You need to say

$$s' = s - \{x\} \wedge x \in s$$

since in the first case x may not be a member of s and the post-condition could hold by returning an arbitrary value; moreover, the set would not change in value, probably not the intended behavior for a *remove* operation.
Saying

$$s - s' = \{x\}$$

is also not strong enough. Here you need to add that s' is a proper subset of s:

$$s - s' = \{x\} \wedge s' \subset s$$

since the first case allows s' to have extra elements.
Saying something like

$$f(choose(s)) \wedge g(choose(s))$$

in the post-condition of an operation where *choose* is not deterministic (i.e., not a function) is weaker than saying

$$\exists\, x \,.\, x = choose(s) \wedge f(x) \wedge g(x)$$

since in the first case the different occurrences of *choose* could return different values. Of course if *choose* is a function, then it is guaranteed to return the same value.
Recursive definitions, commonly found in algebraic specifications, may at first look puzzling. For example, in specifying the *delete* operation for sets

$$\begin{aligned} delete(insert(s, e_1), e_2) = {} & \text{if } e_1 = e_2 \text{ then } delete(s, e_2) \\ & \text{else } insert(delete(s, e_2), e_1) \end{aligned}$$

a common error is to forget to reapply *delete* recursively if e_1 and e_2 are equal or to forget to "reinsert" e_1 if they are not.

6 Challenges for the Technical Community

There exist expressive enough logics to support most kinds of properties that software engineers want to state of their systems, including temporal behavior (safety and liveness properties), real-time behavior (hard and soft time constraints), and erroneous behavior (failures, exceptions, etc.). So we are not lacking in the number and kinds of logics.
However, since no one logic is going to be the most appropriate for specifying all aspects of a software system, one technical challenge for the mathematical logic community is to be able to use different logics together. We might combine logics at the surface (syntactic) level or at the deeper semantic level or at some level in between. We might translate different logics into a common logic.

A more pragmatic need is better and more sophisticated tools to help in the semantic analysis of specifications. As tools increase in logical power, we can do more reasoning of more complex and larger systems. For example, rewrite rule engines and similar tools that manipulate algebraic (equational) specifications can only go so far; since more sophisticated logics are needed to describe real systems, more sophisticated reasoning tools are needed. Analogous to the need for ways to combine logics is the need to combine tools that support different logics and/or different proof techniques. For example, investigating the combination of model checking and theorem proving seems particularly promising and fruitful. Finally, considering the subject of this paper, we still face the educational challenge of teaching mathematical foundations like logic and discrete mathematics to practicing or aspiring software engineers. We need to go beyond giving the traditional courses and think about who the target students are (usually computer scientists and engineers); these new kinds of courses must make direct connections between mathematical concepts and real software systems.

Acknowledgments

I thank David Garlan and Daniel Jackson for their feedback on an earlier draft of this paper, for the fun discussions we have had on this topic, and for their collaboration on designing and teaching the Models, Methods, and Analysis courses for the MSE program.
I thank all the students with whom I have worked over the years for their interest and patience in trying their hand at formal specification.

References

1. DIS 8807. Information systems processing–open systems interconnection–lotos. Technical report, International Standards Organization, 1987.
2. H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.
3. W.R. Bevier, W.A. Hunt, Jr., J S. Moore, and W.D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5:411–428, 1989.
4. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

5. D. Garlan. Preconditions for understanding. In *Proceedings of the Sixth Int'l Conf. on Software Specification and Design*, pages 242–245, October 1991.
6. D. Garlan, G. Abowd, D. Jackson, J. Tomayko, and J.M. Wing. The CMU Master of Software Engineering Core Curriculum. In *Proceedings of the Eighth SEI Conference on Software Engineering Education (CSEE)*. Springer-Verlag, 1995.
7. D. Garlan, A. Brown, D. Jackson, J. Tomayko, and J. Wing. The CMU Masters in Software Engineering Core Curriculum. Technical Report CMU-CS-93-180, Carnegie Mellon Computer Science Department, August 1993.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
9. J.J. Horning, J.V. with S.J. Garland Guttag, K.D. Jones, A. Modet, and J.M. Wing. *Larch : Languages and Tools for Formal Specification*. Springer-Verlag, New York, 1993.
10. C.B. Jones. *Systematic Software Development Using VDM*, chapter 15. Prentice-Hall International, 1986.
11. Deepak Kapur. Towards a theory of abstract data types. Technical Report 237, MIT LCS, June 1980. Ph.D. Thesis.
12. B. Lampson, W. Weihl, and U. Maheshwari. Principles of computer systems. Technical Report MIT/LCS/RSS-22, MIT Lab. for Comp. Science, 1993. Lecture Notes for 6.826, Fall 1992.
13. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. McGraw-Hill/MIT Press, 1986.
14. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. Technical report, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.
15. A.J.R.G. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.
16. L. Mummert, J.M. Wing, and M. Satyanarayanan. Using belief to reason about cache coherence. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 71–80, August 1994. Also CMU-CS-94-151, May 1994.
17. Scott M. Nettles and James W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*. ACM, June 1993.
18. John Scheid and Steven Holtsberg. Ina Jo specification language reference manual. Technical Report TM-6021/001/06, Paramax Systems Corporation, A Unisys Company, June 1992.
19. J.M. Spivey. *Introducing Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.