Computer Science Technical Report

An Abstract State Machine Primer

by James K. Huggins and Charles Wallace

Michigan Technological University
Computer Science Technical Report
CS-TR-02-04
December 4, 2002

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

An Abstract State Machine Primer*

James K. Huggins
Computer Science Program
Kettering University
Flint, Michigan, USA
jhuggins@kettering.edu

Charles Wallace
Computer Science Department
Michigan Technological University
Houghton, Michigan, USA
wallace@mtu.edu

> [1]**prim·er** (′prim·ər, *chiefly British* ′prī·mər) *noun*
> **1:** a small book for teaching children to read
> **2:** a small introductory book on a subject
> [Middle English, from Medieval Latin *primarium*, from Late Latin, neuter of *primarius* primary]
> Source: *Merriam-Webster's Collegiate Dictionary*, Tenth Edition, 1998.

Over the last twenty years, *Abstract State Machines (ASMs)* have been used to describe and validate a wide variety of computing systems [9]. Numerous introductions to ASMs have been written (see for example [20, 10, 21, 36]). In addition, many ASM papers include brief overviews for the casual reader. In this paper we attempt to provide a gentler introduction, focusing more on the use of the technique than on formal definitions.[1] Our target audience is a senior-level undergraduate class in software engineering, but we hope that this work proves helpful to other readers as well.

While there are references to "software" throughout, the reader should be aware that ASMs are relevant to other kinds of system development as well. For simplicity, our focus is on ASMs with only a single agent; distributed ASMs may be considered in a later paper.

We were sitting in a coffee shop when Questor, a bright undergraduate student, stopped by. He mentioned that he was studying ASMs in his software engineering course. Curious, we asked him to sit with us and tell us about his progress.

Questor: It's a little rough. I can follow the basic definitions [21], but I'm finding it difficult to use those definitions to actually write an ASM. How did *you* learn?
Author: Well, I suppose it was through a sort of apprenticeship process. Fortunately for us, we had the originator of ASMs as our mentor. We started by looking at ASMs that he and others had created; then we tried making our own. Through an iterative process of criticism and revision, we honed our ASM-writing skills. We've also tried out our ASMs by executing them, using the many tools available [4, 38, 3, 2].
Q: It seems like there ought to be a way to pass some of your experience on to new ASM writers like me.
A: Let's give it a try.

ASMs and the Software Process

Q: So how do you go about using the ASM method?
A: In some ways, it's not all that different from other methods you may have used in developing software.

*Financial support for this work has been provided by Microsoft Research.

Q: But I thought that this was supposed to be a totally new and different technique. If the process isn't all that different, why bother with it?
A: Maybe you have an exaggerated view of what ASMs are all about. First of all, let's make it clear what the ASM method *isn't*. It doesn't define the entire software *process*; it doesn't tell you the steps to take to develop a product. You still need to solve the same old problems: how to find out what your customer wants, how to design the product carefully, how to check that what you're producing is what the customer wants. Many people have proposed so-called *process models*, that give you guidelines on how to attack these problems. The ASM method isn't a process model.
Q: I've heard of some process models, like the waterfall model [33] and Extreme Programming [5]. So if the ASM method isn't a process model, what is it?
A: At its core, it is simply a method for documenting software at various stages of the development process. The goal is software documentation that is clear and precise. Of course, the ASM method doesn't *guarantee* clarity or precision; that's still up to you.
Q: So this is a new way of documenting software. What's so exciting about software documentation?
A: When we think of "documentation", we tend to think of those thousand-page reference manuals that no one ever reads. But documentation doesn't have to be like that. It can be succinct, elegant, and informative. And it *ought* to be. Documentation is the medium through which different players in the development process communicate with one another. If the documentation is inaccurate or unintelligible, the software process breaks down.
Q: So why is ASM better than other means of documentation? For instance, what's wrong with nicely commented C or Java code?
A: Nothing at all; well-commented code is a wonderful thing. And certainly your software will eventually be expressed as a program in a language like C or Java. In fact, you can view an ASM as a program, though the language of ASMs is different from traditional programming languages in many ways. The most important difference between an ASM and a "conventional" program is that an ASM can represent software at a much higher level of abstraction.
Q: Why would I care about such a description?
A: You can produce this high-level description at a much earlier point in your design process, before all the details of the design have been decided. When writing a program in a traditional programming language, you are often forced to make design decisions that may be premature.
Q: So once I decide on all the details, I can forget about ASMs and just use my favorite programming language?
A: Even then, such a description might be too detailed for most people to understand. Suppose you've never written in programming language L before. Notice that a compiler for L is in essence a complete description of L; it defines how instructions in L should be interpreted and executed. But would you want to learn how to write programs in L by studying the source code for the compiler?
Q: No — it would be way too complicated.
A: Too much detail can obscure what's important. Through abstraction, you can produce a view of your software that omits those details you don't want to see at a given time.
Q: Let's go back a bit. Earlier you said that you can write an ASM before all the details of your design have been finalized. If you're not sure about all the details of a design, why not just describe the design using a diagram, or good old-fashioned natural language?
A: The ASM method isn't meant to replace these approaches. But as we all know, diagrams and natural language can be vague or ambiguous. This is good in art and literature; we can find many "meanings" in a painting by Picasso or a sonnet by Shakespeare. And everyone's favorite humor form — the pun — would be impossible without the ambiguity of natural language. But in a description of a software system, ambiguity can be a very bad thing [26], particularly when the description is shared among many people. Unlike natural language, the ASM method has precise semantics; there's never any doubt about the meaning of an ASM.
Q: It seems like abstraction is an important benefit of the ASM method; I suppose that's why the word "abstract" is in the name. Are there other advantages?

A: Certainly. ASMs can be executed directly using various tools; this makes the transitions from designing to coding, and from designing to testing, much easier. In our opinion, the notation used to write an ASM involves minimal overhead and is easier to learn and use than those of other methodologies. There are other benefits as well.
Q: OK. So we'll see them along the way. I guess I can trust you for now.
A: Why the suspicion?
Q: It just seems like from what I've been hearing, ASMs were supposed to solve everything and turn all of my hard problems into easy problems.
A: No method, ASMs included, could possibly fulfill such a claim. It's a logical impossibility.
Q: What do you mean?
A: Suppose that you could use ASMs to turn a hard problem into an easy one. Then the original problem really wasn't hard to begin with, was it?
Q: I suppose not; if the ASM solution is easy, then the problem itself must be easy. This sounds a lot like the notion of reducibility from complexity theory [35].
A: Exactly. Designing computing systems is inherently difficult; the complexity of computing systems is an essential property of such systems, not an accidental one [14].
Q: I suppose the best we can hope for is that using the ASM method doesn't make a hard problem even harder.
A: Correct. But that's actually a great achievement. Part of the essential difficulty of software development is that the simple ideas behind algorithms get obscured by encoding them in a language. If the original ideas, be they simple or difficult, can be expressed at the same level of complexity using ASMs, we have achieved a good thing.

Requirements and Specifications

Q: Very well. So, how shall we proceed?
A: We'll start by considering the requirements of the system we're modeling or designing. From there, we'll proceed to analysis, design, and finally verification.
Q: This sounds a lot like the waterfall process model.
A: Yes, we're intentionally using that model here.
Q: Why are you choosing the waterfall model? You did say earlier that ASMs can be used with any process model — so why not use something newer, like Extreme Programming, or the spiral model [7]?
A: We'll use the waterfall model here to allow us to focus on various aspects of writing ASMs in a "stand-alone" manner. But one often does a number of these activities at the same time.
Q: OK. So, on to requirements.
A: Our job as software developers, in very simple terms, is to write a program for a machine. Our customers want this machine to accomplish something in "the world", which we'll refer to as the machine's *environment*. Customers draw up a set of requirements, which are statements that define what the software must do: the effects it must have on its environment.
Q: Not just what it must do — what it must *not* do, as well.
A: Good point.
Q: From what I remember, requirements are not supposed to say anything about *how* the software actually works.
A: Right. In other words, requirements refer solely to the environment.
Q: Requirements *never* refer to the machine? That seems odd.
A: Well. . . there's a certain amount of overlap between the environment and the machine. Some aspects of the machine are purely internal to the machine, while some parts of the environment are completely external to the machine. But there is also an *interface* between the machine and its environment. For instance, a program instruction may activate an output device. This output device then affects the environment in some

way — for instance, by printing an image, by sending a message to another computer over a network, or by issuing $20 bills from an ATM.
Q: In the other direction, actions in the environment may affect the machine — for instance, a user or another computer may send signals that cause the program to do something.
A: That's right. Now, the conditions that the customer places on the environment are called requirements. Of these requirements, the ones that pertain to the environment-machine interface form the *specification*.
Q: And since we software developers are responsible for the machine, we should concentrate on the specification.
A: Exactly — though we should always keep in mind the "big picture", the full set of the customer's requirements.
Q: Now, how about conditions internal to the machine, with no relation to the environment. When do we consider those?
A: Since they are purely internal to the machine, the customer doesn't know or care about them. They are considered only during the design stage.
Q: I see the point. Sometimes this distinction between what's in the machine — and under our control — and what's in the environment — and out of our control — is tricky.
A: Indeed. For example, imagine you're creating software to control an elevator. The customer would probably insist that the elevator car only stop at points where there is a floor — not between floors two and three, for instance. Would that requirement be part of the specification?
Q: If we're just writing the software, then the elevator car is actually just part of the environment. Our software simply sends signals that — hopefully — translate into correct actions by the elevator. So, I guess that requirement wouldn't be part of the specification.
A: Good. We software developers don't have any control over the elevator car itself; that's the domain of mechanical engineers. So we can't really guarantee anything about its behavior.
Q: Our software may conform perfectly to the specification, and the elevator could still do bad things: the mechanical controls might not be hooked up to our software controller, the power might be disconnected, the elevator car might be physically jammed, and so on.
A: Yes. This is especially important when the environment contains other software running on other machines. Say we're designing a module that communicates with other modules. We mustn't incorporate aspects of the other modules into our specification — even though sometimes it might be tempting to do so.
Q: So what does all of this have to do with writing an ASM?
A: An ASM is simply a program for a machine — an abstract one. When we write this program, we must be careful to distinguish phenomena that are under our control from those that are strictly part of the environment. Our program must not try to control aspects of the environment that are outside the specification.

On the other hand, we must be careful not to add too many details about the machine initially. If we want to be *really* careful, we can write an ASM that describes the behavior at the machine-environment interface — and nothing else. Anything that happens in the environment or the machine outside of this interface is left unexplained for the time being.
Q: That ASM would be a specification.
A: Right. Regardless of where we start, we continue by writing a series of ASMs that gradually add machine-internal details, up to the point where we have a "fully implemented" version. It's up to us to decide where that point is.
Q: How do you write a *program* that leaves internal details unexplained? That seems weird.
A: We'll see. Let's start talking about how to write an ASM.

Analysis: ASM Universes and Functions

Q: OK — where do we start?

A: A good first step is to determine the kinds of data that our software will manipulate, the means by which we can manipulate that data, and how we'll store the results of our computation along the way.
Q: By "kinds" of data, don't you mean "types"?
A: Well, we have a pretty simple notion in mind: a collection of data elements on which certain operations can be performed. We avoid the word "type" because ASMs are not typed in the sense of typed programming languages. For instance, there are no guarantees that an ASM is type-safe, and the ASM method doesn't define how to create complex types from simpler ones. Furthermore, you can move an element in and out of one of these collections during execution, but you cannot do that with types.
Q: Well, then aren't these kinds just sets?
A: Yes. For historical reasons we call these sets "universes"; the term comes from the world of mathematical logic. Calling them universes instead of sets allows us to think about sets as data values themselves with less notational confusion.
Q: It does get confusing to have to think about sets of sets.
A: Exactly. So, one of the first things to consider at this stage is the universes needed by a particular application.
Q: Can we discuss some examples?
A: Sure.

Greatest Common Divisor

A: Let's consider the problem of computing the greatest common divisor of two positive integers. What universes would we need?
Q: Well, since you mentioned the words "positive integers", it seems like we would need the set of positive integers.
A: Very good. Often, the statement of the problem gives clues as to the types of data values needed. Let's give this universe the name PosInt.
Q: It occurs to me that the set of positive integers is infinite. Doesn't that present a problem?
A: Not at this point. We're designing the "ideal" algorithm here, and the infinite set of positive integers is the appropriate set.
Q: But won't this pose problems if you want to execute this on a computer later?
A: Possibly. But that's an implementation issue rather than a design issue. We should focus on getting the design right. If necessary, we can revisit this decision later once we know the details of the target implementation.
Q: But wouldn't it be prudent to make those decisions now, so that we can design our algorithms with this in mind?
A: Paradoxically, that usually ends up making things worse rather than better. Adding those details early often obscures the essence of the algorithm.
Q: OK. So we have the set of positive integers. Is there anything else that we need?
A: Well, we might want a set of "modes" to help us keep track of what we're doing, kind of like an instruction counter in assembly language.
Q: We could just use the positive integers for that, couldn't we?
A: Sure. It might make things clearer to use a different set, though. But let's visit this question later.
Q: What's next?
A: Well, here's one possibility: simply define a function gcd that takes two PosInts a and b and returns the PosInt that is the gcd of the two arguments: a number that divides a, divides b, and is divisible by any c that divides a and b.
Q: Come on. That's cheating. There's no algorithm there; you've just assumed the existence of a function that computes the answer! How do I even know that your gcd function is computable?
A: This is obviously a very high level of abstraction. For our simple problem, it seems a bit silly. But think about what such an ASM gives us. It describes exactly the input-output behavior of the machine, without any details about the internals of the machine. What does that sound like?

Q: A specification of the problem.
A: Right. Now, it turns out that the specification of this problem is trivial. But many real-world problems are so complex that even getting the high-level specification right is not easy.
Q: OK. But I'd still like to see an example where such a high level of abstraction is really useful.
A: That's a good idea for later.[2] We'll stick to simpler examples for now.
Q: All right. Can we create a more interesting ASM for the gcd problem?
A: Yes, but we'll have to make some design decisions. What algorithm do you have in mind to compute the gcd?
Q: I remember Euclid's algorithm [29].
A: What operations does that algorithm use?
Q: It uses the modulus, or remainder, operation, as I recall. So I'm guessing we'll need that operation.
A: Fine. So we'll include a function mod that takes two PosInts and returns the modulus of those two values. In keeping with mathematical tradition, we'll use mod in infix form — in other words, $a \,\mathsf{mod}\, b$ rather than $\mathsf{mod}(a,\ b)$.
Q: Wait a minute. It occurs to me that this function could return zero as a value, and zero isn't a positive integer.
A: Good point. So it looks like we'll need the set of natural numbers, which we'll call Nat. We'll assert that PosInt is a subset of Nat. Let's also add a zero-argument function 0 to name that special value zero that's in Nat but not in PosInt.
Q: Then the function mod takes two PosInts and returns a Nat.
A: Correct. We'll sometimes write this as $\mathsf{PosInt} \times \mathsf{PosInt} \rightarrow \mathsf{Nat}$, using the conventional mathematical notation for functions.
Q: What else do we need?
A: We'll probably need the equality relation defined on natural numbers. We'll define it as a function that takes two Nats and returns a Boolean — in other words, $\mathsf{Nat} \times \mathsf{Nat} \rightarrow \mathsf{Boolean}$.
Q: Hold it — What's Boolean? I don't think we've defined it yet.
A: You're right. Boolean is simply the universe of Boolean values, comprising the two distinct values true and false. This universe is so commonly used in ASMs that we usually don't bother naming it directly.
Q: Couldn't we do without the Boolean universe? There is no Boolean type in C, for example.
A: How does C deal with logical operations?
Q: Numerical values of some integer type are used instead. Zero is interpreted as "false"; anything else is interpreted as "true". We could do the same thing here: let the Equal function take two Nats and return 1 exactly when the two arguments are identical.
A: This is probably too low a level of abstraction for the problem at this point. Do you think it would be more natural to see Equal(a,b)=1 or Equal(a,b)=true?
Q: The latter, I guess. Though I'd probably rather see a=b.
A: True; and that's how we'll usually write it.
Q: Isn't that cheating?
A: We prefer to call it "syntactic sugar". We can view a=b as an abbreviation for Equal(a,b), even if there isn't much savings in the length of the text. This is pretty standard practice in programming languages. For example, contrary to popular opinion, C doesn't have arrays; the array notation in C — square brackets and all — is just syntactic sugar for pointer arithmetic [28].
Q: Really? I'll have to look at that.
A: That's the power of abstraction. You can view C as a language with arrays when it's convenient; if you need to see C at a lower level of abstraction, you can do that as well.
Q: Could we consider the gcd problem at other levels of abstraction as well?
A: Sure. Can you imagine a lower level of abstraction?
Q: I suppose someone might complain that the mod function is too high-level, since it can be defined in terms of lower-level operations such as addition and subtraction.
A: Right. So we could define the usual addition and subtraction functions and only use those during the coding phase. But for now, we'll stick with the mid-level abstraction functions you mentioned.

Q: What's left?
A: Storage space for our intermediate results. What will we need?
Q: Well, we'll certainly need places to store our two input values and our desired output value.
A: Right. So, let's define nullary functions A, B, and Output to hold PosInts during our computation.
Q: What do you mean by "nullary function"?
A: Literally, a function that takes no arguments and returns a value. Think of it like a variable in your favorite programming language.
Q: But variables usually change during the course of a program, while I usually think of functions as artifacts that never change.
A: Again, it depends on your point of view. One way of seeing a function is as a mapping from input values to output values. We simply choose to change the values of that mapping over time.
Q: So, a one-argument function is really like a one-dimensional array, where a function call is comparable to array lookup and a function update is comparable to storing a new value at a location in an array.
A: Exactly. In the same way, multi-argument functions can be seen as multi-dimensional arrays.
Q: I see. So, now we have all the universes and functions defined.[3] Can we start writing the program?
A: We can — but before we do, let's practice defining universes and functions for a few more examples.

String Matching

Q: OK. What should we look at next?
A: Let's consider the problem of string matching: finding the occurrence of a string within another string.
Q: What should happen if there is no such occurrence — or if there are multiple occurrences?
A: These are important questions that should be answered through requirements analysis. For the moment, let's put as few restrictions on the problem as we can. Let's use "haystack" to refer to the string to search, and "needle" to refer to the string to search for. We'll assert that the needle might not appear at all in the haystack, and if multiple occurrences are present, our algorithm may identify any one of them. If the needle doesn't appear in the haystack, we'll return a special value undef, for "undefined".
Q: You're also assuming that the needle is no longer than the haystack, right?
A: Good question. Let's not make that assumption — though it certainly must be the case in order for there to be a positive answer. Depending on how we write the algorithm, we might decide to perform an initial length comparison between needle and haystack, terminating immediately if the needle is longer.
Q: OK. So, we start with universes. We'll obviously need universes of characters, arrays, and positive integers to access positions in the arrays.
A: Why do you need arrays?
Q: Strings are arrays of characters, aren't they?
A: Are you sure?
Q: In C, strings are represented as arrays of characters in memory, terminated by a special value.
A: But is that true for every language? In Java, for example, strings are objects whose implementation isn't accessible to the programmer.
Q: I guess I've made my universes too low-level, haven't I?
A: It all depends on the algorithm we implement. But arrays may be too system-specific, and thus too low-level, for our purposes.
Q: OK. Let's try again. We'll certainly need a String universe, since strings form the input. What are we returning as output?
A: A good question; there are several possibilities. We could simply return true or false, according to whether the needle is found. We could return the substring of the haystack that begins with the needle and contains the rest of the string. Or, we could return the position of the occurrence of the needle in the haystack. Let's use the latter.
Q: We'll probably need some sort of counting numbers — integers, natural numbers, or positive integers — for that purpose. Will we start counting positions in the string from zero or one?
A: Another good question. Let's start counting from zero.

Q: Fine. So we'll use String and Nat. We still might need a Character universe, since strings are composed of characters.
A: That seems to be natural enough. One could argue that characters are just strings of length one, but that seems to be less natural.
Q: Agreed. Let's consider functions. If we wanted to be extremely high-level, as before, we could just assume the existence of a function Find: String $\times$ String $\rightarrow$ Nat and be done.
A: Once again, you'd have a fully abstract specification of the problem. But let's move on to a lower level of abstraction.
Q: Right. We'll probably need a substring operator; it should have the form Substr: String $\times$ Nat$\times$ Nat $\rightarrow$ String.
A: What do the second and third arguments signify? I can imagine at least a couple of interpretations.
Q: The second argument indicates the starting position; the third argument indicates the length of the substring.[4] That reminds me; we'll probably need a function Length: String $\rightarrow$ Nat.
A: Good. Anything else?
Q: We'll probably need some temporary storage variables, or nullary functions, as you call them. We can always come back and add more universes and functions if we need them, right?
A: Sure. Do you always know all the variables you need when you sit down to write a program?
Q: No. It's nice to know that this method is flexible enough to handle iterative development like this.
A: Absolutely.
Q: By the way, we said that we needed a universe of characters, but we haven't identified any functions that use them, either as input or output.
A: Good observation. Perhaps we won't need them after all. We'll see how things progress.

Minimum Spanning Trees

Q: Let's try one more example.
A: Sure. Consider the problem of finding a minimum spanning tree in a graph. Do you remember the definitions?
Q: Let's see. A *graph* is a set of *nodes* and a set of *edges* connecting pairs of nodes. In a *weighted* graph, each edge also has a numerical weight. A *tree* is a connected acyclic graph. A *spanning tree* of a graph is a subgraph tree that includes all nodes in the graph. A spanning tree of a weighted graph is a *minimum* spanning tree if its weight — the total weight of all edges in the tree — is the minimum over all possible spanning trees for that graph.
A: Right. Sometimes the problem is phrased in terms of finding just the weight of such a tree, but typically this is solved just by finding an actual tree and computing its weight.
Q: OK. We obviously need a way to represent graphs. I know that there are two common ways for graphs to be represented as data structures: adjacency lists and adjacency matrices. But I'm guessing that this is too low-level for us right now.
A: Why?
Q: Well, I was taught that the reason for choosing one of these representations over the other is performance, not correctness. At this point, we haven't decided if we're interested in performance analysis as opposed to algorithm correctness.
A: Good answer. So what would be the most general way of representing our graphs?
Q: Probably as sets. So we'll have a universe of Nodes and a universe of Edges, and a function Endpoints that maps each Edge to the set of nodes that it connects.
A: You're implicitly assuming an undirected graph here, aren't you?
Q: I suppose that I am. But I seem to recall that the problem isn't all that different for directed graphs, so we could always produce another version for directed graphs.
A: Right. In addition, we usually assume that there is at most one edge between any two points.
Q: Good. Since we have to deal with weighted graphs, we'll need a set of Weights, which is probably just the set of non-negative real numbers, and a function Weight: Edge $\rightarrow$ Weight.
A: Why not allow negative weights?

Q: I suppose we could, though I seem to recall that some algorithms on graphs don't work if you have negative weights.
A: True enough. Let's stick with non-negative weights for now.[5]
Q: OK. So that should be enough to represent the graph.
A: Before we continue, can you think of an alternative way of representing the graph? I'm thinking of a way that doesn't require an Edge universe.
Q: Without an Edge universe, how will you know which Nodes are connected by edges?
A: How about a binary function defined on Nodes?
Q: OK, we could define a function Edge that takes two Nodes and returns true or false, depending on whether there's an edge between them.
A: Yes, that seems fine. But for an *undirected* graph, we'd have to assert a condition on the Edge function. What condition?
Q: Well — the order of the points shouldn't matter. So if $\mathsf{Edge}(p, q) = \mathsf{true}$, then $\mathsf{Edge}(q, p)$ should also be true.
A: Right. Now, how about weights?
Q: We could define another function Weight: $\mathsf{Node} \times \mathsf{Node} \rightarrow \mathsf{Weight}$ that gives you the weight of the edge from one point to another. But wait — what if there's no edge between Nodes p and q? What should $\mathsf{Weight}(p, q)$ return?
A: We could add a special value — call it Infinity — to the Weight universe. Infinity would be greater than any other Weight.
Q: Now that I think of it, we wouldn't even need the Edge function. To check for the presence of an edge between p and q, we could just check whether $\mathsf{Weight}(p, q) = \mathsf{Infinity}$.
A: Good point.
Q: It seems like this approach is better than my original one. We only need two universes instead of three.
A: Well, it's not so obvious that this approach is better. That really depends on other aspects of the problem. For instance, this approach doesn't allow us to represent multiple edges between two points. That's not a problem with a universe of Edges. For now, let's stick with the original approach — and we'll continue to assume that at most one Edge connects any two Nodes.[6]
Q: All right. Well — at this point, I'm not sure how to proceed.
A: What's the problem?
Q: I can remember several algorithms for solving this problem that use different data structures: queues, stacks, union-find forests of trees, and so on. The data structures we need will depend on how we choose to implement the algorithm.
A: True. So perhaps we should defer working on this part of the design for a while.
Q: It seems like these phases can't be so easily separated after all. In our design work, we keep finding questions that make us look back to requirements and others that make us look forward to coding issues.
A: Just as it happens in real-life software projects.
Q: I see what you mean. But I begin to see the point; to write a "good" ASM, I need to think about higher levels of abstraction than I usually think about.
A: Exactly. Most of us have written in programming languages like C and Java for so long that we've come to feel like our favorite language library is the only way to think about representing data. But with abstraction, things can become much cleaner and clearer.
Q: Is that why some of my algorithms textbooks tend to have so many of their algorithms written in pseudo-code?
A: That's certainly one good reason. Sometimes, writing an algorithm in pseudo-code makes it much clearer than when all the details of a "dirty" programming language have to be included. In fact, ASM programs have been called "pseudo-code over abstract domains" [9], which may indicate why ASMs can be so clear.
Q: It sounds like we're ready to move on to writing an ASM program.
A: Indeed it does.

Design: ASM Rules

Q: To write a program in a language like C or Java, I use various *statements*: conditional statements, loop statements, and so forth. What kinds of ASM statement are there?
A: ASM statements are called *rules.* The most basic rule[7] is the *update rule*, which has the form:

foo(t1, t2,... tn) := t0

Here, foo is an n-argument function, and t0 through tn are *terms*, or expressions. Executing the rule updates the value of the function foo at the specified arguments to the specified value.
Q: The := operator is an assignment operator, like = in C and Java?
A: Yes, they're similar. But note that = in C and Java is actually an expression, which returns a value; it not only performs assignment but also returns the value assigned. An ASM update rule doesn't return anything; it just performs an update. An unfortunate consequence of the C and Java notation is that assignment and equality are often confused, since the standard mathematical symbol for equality is =. We use := in update rules to avoid this confusion. In ASM, = means equality, not assignment. Incidentally, other programming languages such as Pascal use := for assignment.
Q: I'm still a little uncomfortable with talk about changing the meaning of functions; I guess I still have this picture in mind of a "black box" that remains constant. But we mentioned earlier that functions are similar to arrays. I guess if we wrote

foo[exp1, exp2,... expn] := exp0

instead of the rule above, I wouldn't have a problem.
A: The meaning is as you suggest. Continuing, we have the *conditional rule*, which has the form

if cond **then** *rule1* **else** *rule2* **endif**

The meaning of the rule is the obvious one.
Q: Let me make sure I understand it. cond is presumably a term that evaluates to a Boolean value. If the value of cond is true, *rule1* is executed; if the value of cond is false, *rule2* is executed.
A: Correct. We frequently use variations on the rule above such as

if cond **then** *rule1* **endif**

or

if cond1 **then** *rule1*
elseif cond2 **then** *rule2*
...
elseif condn **then** *rulen*
else *rule0*
endif

with the obvious meanings.
Q: Those rules can be defined in terms of the basic conditional rule.
A: Correct. We use the abbreviated forms above to improve readability.
Next, we have the *block rule*:

do-inparallel
rule1
rule2
...
rulen
enddo

To execute a rule of this form, execute all of *rule1* through *rulen* simultaneously. Each subrule can be any kind of rule, including a block rule.

Q: "Simultaneously", eh? I guess this would be useful if you're designing software for some sort of parallel machine.

A: Well, possibly. But it turns out to be very useful in designing traditional sequential software.

Q: How? When I write a sequential program, the instructions are executed sequentially, not in parallel.

A: I think part of the issue is that much of the sequentiality that we see in real-life programming isn't essential, but an accidental artifact of the languages we've chosen to use.

Q: What do you mean by "essential" rather than "accidental"?

A: Consider the problem of swapping the contents of two variables. How would you accomplish this in most programming languages?

Q: Typically, to swap the values of `foo` and `bar`, we would write three consecutive lines of code like this:

```
temp := foo; foo := bar; bar := temp;
```

We need `temp` to hold one of the values temporarily, so that the other value can be placed in its old location.

A: Why do you put the value of `foo` into `temp`? Wouldn't it be just as correct to write

```
temp := bar; bar := foo; foo := temp;
```

Q: Sure. The result is the same: the values are swapped.

A: Then it really doesn't matter whether `foo` or `bar` is placed into `temp`.

Q: Right.

A: Notice that this isn't immediately clear from your code. A casual observer might not see that the choice of `foo` or `bar` at that point is arbitrary.

Q: I see. I suppose that it wouldn't be a problem in such a simple piece of code as this, but it would be easy to write far more complicated code in which the sequentiality relationships aren't obvious at all.

A: Consider in contrast the corresponding ASM rule:

do-inparallel
foo := bar
bar := foo
enddo

Notice the independent relationship of `foo` and `bar` above. Neither variable has priority over the other; both are equally important.[8]

By the way, we often omit the keywords **do-inparallel** and **enddo** above when their meaning is clear from context. So we would sometimes write the above simply as

foo := bar
bar := foo

or perhaps even

foo := bar, bar := foo

where the comma above is simply a syntactic separator and does not indicate any form of sequentiality.
Q: We seem to be discussing abstractions again. At a high level of abstraction, swapping can be seen as a simultaneous, parallel operation. And perhaps this is a more natural level of abstraction; I seem to recall as a beginning programming student making the mistake of writing

```
foo := bar; bar := foo;
```

and having to be trained to think in non-parallel terms.
A: Quite right. Of course, the notion of only performing one operation in a single step is violated all the time in computing.
Q: Do you have examples?
A: Think of hardware circuits, in which multiple operations occur during a single "step", or "clock cycle", if you prefer. Or think of assembly language programs, in which performing an arithmetic operation involves not just storing a value in a register but also setting various control bits.
Q: I suppose there is more parallelism in so-called "sequential" computing than I had considered. There must be a limit to it somewhere, right?
A: Certainly. Otherwise, we could accomplish an unbounded amount of work in a single step, which violates our intuitions as to what "sequential algorithm" means. The detailed explanation is somewhat subtle [23], but the essential idea is that only a bounded amount of work should be accomplished at each step.
Q: So doing multiple things at once is fine, as long as we never do more than, say, seventeen things at once — or some other fixed number.
A: Correct.
Q: But what do you do when sequentiality *is* essential to part of an algorithm? Often, results of early computations are used in later computations in an algorithm; clearly, the earlier computations need to complete before the later computations can begin.
A: The effect of sequentiality can be modeled by means of conditional rules which only allow the later computation to proceed once the earlier computation has finished. Consider the following:

if not Initialized **then**
    Initialize
    Initialized := true
else *Main*
endif

Here, *Initialize* is a rule that sets up initial values for the "real" computation, as represented by the rule *Main*. Initialized is simply a nullary Boolean function whose initial value is false.
Q: Let me see if I have the right idea. In the beginning, Initialized = false, so when the conditional rule executes, only the updates in the **then** clause take effect. That's one step. Then for the next step, the whole conditional rule is executed again, but this time Initialized = true, so only the updates in the **else** clause take effect.
A: Exactly.
Q: I get it, but it's sort of unusual. When I execute a C program, for instance, at any given moment a particular statement is being executed. I can point to a statement in the source code and say, "That's the statement that's currently being executed". In an ASM, there's no notion of a particular subrule within the main rule as being the "current" one. The *whole* rule is executed at every step.
A: That's right. Of course, you could simulate the notion of a "current rule" by introducing a nullary function CurrentRule. Then your ASM would test CurrentRule and execute the corresponding rule:

if CurrentRule = 0 **then** *Rule0*, CurrentRule := 1
elseif CurrentRule = 1 **then** *Rule1*, CurrentRule := 2
⋮

Q: That certainly seems correct. But this seems to be a lot of additional work to do in order to preserve your parallel block rule.
A: If you find that sequentiality is really necessary to your algorithm, and that the techniques we've described above obscure the algorithm rather than illuminate it, there are language constructs for ASMs that add explicit sequential composition [11]. But those constructs can all be defined in terms of the rules we presented.[9]
Q: Have we seen all the basic rule types?
A: There are a few others; we'll introduce them later, when we have a need for them. The rule types we have shown are sufficient for all sequential algorithms [23].
Q: It occurs to me that we haven't described what a computation is.
A: We need to present the notion of a *run*.[10] A *program* is simply a rule. A *run* is a sequence of *states*, where each state is obtained from the previous one by executing one step of the program in the previous state.
Q: And what is a state?
A: Informally, a state is the collection of functions and universes we defined earlier, along with their current values at that moment in the computation.
Q: So, a state is like a "snapshot" of the memory of the computer running the algorithm at a given moment during its computation.
A: Correct. Of course, we're usually looking at a much higher-level notion than bits in memory, but the intuition is right.
Q: I think I've seen enough definitions for a while. It's time to return to our examples.

Greatest Common Divisor

A: Let's go back to the greatest common divisor problem. We decided to use Euclid's algorithm. How does it work?
Q: If I recall correctly, Euclid's algorithm relies on the mathematical identity

$$gcd(a, b) = gcd(b, a \bmod b)$$

provided that $a \bmod b \neq 0$. If $a \bmod b = 0$, then clearly b divides a and $gcd(a, b) = b$.
A: Correct. So how does Euclid's algorithm use this identity?
Q: To find $gcd(a, b)$, we first check and see if $a \bmod b = 0$; if so, we can report b as the answer and terminate the algorithm. Otherwise, we recursively calculate $gcd(b, a \bmod b)$.
A: Here's a first attempt at a program for Euclid's Algorithm:

```
if Output = undef then
    if (A mod B = 0) then Output := B
    else
        A := B
        B := A mod B
    endif
endif
```

Q: I see the parallelism of blocks that we discussed earlier is very handy here. We can perform A := B and B := A mod B without worrying about the sequential interactions between the two statements.
A: Precisely. What objections might you have to this program?

Q: Well, you seem to be assuming that A and B are both legitimate values.
A: What do you mean by "legitimate"?
Q: For this to work, A and B both must be positive integers.[11] Further, A must be greater than B.
A: True. We can simply assert that A and B initially have PosInt values.[12] It turns out that the above algorithm does work even when A is less than B.
Q: It does?
A: Trace the execution of the program above for a couple of steps and see what happens. In the meantime, are there other questions about this program?
Q: I'm thinking back to your definition of run that you gave earlier. How long does a run last?
A: A run is simply defined as a sequence of states, and this sequence could be of infinite length.
Q: I usually don't think of Euclid's algorithm as running forever. It should terminate after a finite number of steps.
A: Notice that in this case, once Output is updated, each succeeding state is identical to the previous one. If you want to just consider the initial portion of the run up until that time as the "true" run of the algorithm, that's fine. But that may depend on the application you're considering.
Q: Is it ever useful to have a program that runs forever? Having an infinite loop in my program seems like a bad thing.
A: Well, ASMs are supposed to be able to model all algorithms. If you can write an infinite loop in your favorite programming language, the behavior of that program should correspond to some ASM.
Q: I can understand that argument. But do people ever intentionally write programs that run forever?
A: Of course. How about an operating system? In principle, if the user never explicitly shuts it down, the operating system should never stop. The run of the corresponding algorithm should be an infinite sequence of states.
Q: I hadn't thought of that. But we're getting pretty far from the original algorithm again.
A: True. Let's consider the question of abstraction. Our program shows Euclid's algorithm at its natural level of abstraction.[13] Earlier we talked about other levels of abstraction that we might consider. What would the ASM programs for those levels look like?
Q: Probably the easiest one would be at the highest level of abstraction, where the program would just look like

if Output = undef **then** Output := gcd(A,B) **endif**

or perhaps even

Output := gcd(A,B)

A: What's the difference between the two programs?
Q: In the first program, Output is only updated once with the correct answer. In the second program, Output is updated multiple times, but each time with the same correct answer.
A: Exactly. Whether or not this makes a difference may depend on your application or other matters of interest.
Q: Moving to the other extreme, if we don't have the modulus operation, we have more work to do. I suppose we could create a small ASM that computes the value of A mod B and then use the small ASM inside the larger one.
A: Right. To compute A mod B, start with A and then continually subtract B until you get something less than B.
Q: OK. We'll need somewhere to store the intermediate results.
A: Right. Let's define a nullary function AmodB. Perhaps something like this would do:

```
if CurrentMode = start then
    AmodB := A
    CurrentMode := calculateAmodB
elseif CurrentMode = calculateAmodB then
    if AmodB < B then CurrentMode := doEuclid
    else AmodB := AmodB - B
    endif
elseif CurrentMode = doEuclid then
    if AmodB = 0 then
        Output := B
        CurrentMode := done
    else
        A := B
        B := AmodB
        CurrentMode := start
    endif
endif
```

Note that we've sneaked in some extra functions: subtraction and the less-than relation on natural numbers, as well as the nullary function CurrentMode.

Q: CurrentMode is like the Initialized and CurrentRule variables that you talked about earlier.

A: Exactly. In this case, of course, we move back and forth between the various modes many times until we have our desired answer. It's a common technique in ASMs for enforcing sequentiality. Formally, we define a universe of Modes and define CurrentMode to be a nullary function of that type. From the way we use it here, you can see that for every step where we change A and B in the high-level view of Euclid's algorithm, there are several mini-steps here where we calculate A mod B through repeated subtraction.

Q: I notice that you have four Modes but only rules for three of them.

A: Exactly. After CurrentMode := done is executed, no other rules will be able to execute. This is another way of bringing the computation of the ASM to a close.

Q: What does this universe of Modes look like?

A: Anything you like. The only thing that we require is that it have the four distinct elements named above, and that we can perform comparisons on them.

Q: Why not just use integers? You could write conditions like CurrentMode = 1 instead of CurrentMode = start and eliminate the need for the universe entirely.

A: Perhaps. Earlier, we had mentioned the possibility of eliminating the universe of Modes and using integers instead; the resulting ASM would certainly still compute the correct value. On the other hand, the rules above are probably much more readable with the names attached.

Q: True. I've come to appreciate the value of good names in my programs.

A: In ASMs this is especially true. We've had numerous situations where outside readers were able to help us correct our ASMs, even without a great deal of understanding of the topic area, simply because we used good names.

Q: Getting back to Euclid, this last program certainly seems more awkward than the previous ones. I can see our original ASM in the last clause of the main **if** rule, but it's obscured by all this CurrentMode business.

A: Some of the extra complexity is inevitable, since this ASM does more. In fact, you can view it as the combination of two smaller ASMs. One calculates A mod B and corresponds to the first two clauses of the main **if** rule. The other, corresponding to the last clause, performs Euclid's algorithm, given a proper value for AmodB.

By the way, it turns out that we don't really need CurrentMode at all. If you don't like it, maybe you can figure out how to eliminate it.

Q: OK — I'll try that later.[14] In the meantime, shall we move on?

String Matching

A: Let's look at the string matching example we considered earlier.
Q: Here, I'm not sure how to proceed. I have so many algorithms in mind that I'm getting them confused.
A: Perhaps the confusion might be significant. What algorithms do you have in mind?
Q: The so-called "naïve" brute-force matching algorithm, the Knuth-Morris-Pratt algorithm [30], and the Boyer-Moore algorithm [12].
A: Can you think of any commonalities between them?
Q: All of them end up finding the position of the needle in the haystack, but by very different means.
A: Perhaps we can abstract out that process. Let's consider the following high-level program:

choose i: Nat: Substring(Haystack,i,Length(Needle)) = Needle
    Output := i
ifnone Output := undef
endchoose

Q: What's this "choose" rule?
A: The general form of the rule is

choose x: U: P_x
    $rule_x$
ifnone $rule_0$
endchoose

The idea is as follows. x is a variable that ranges over the universe U. P_x is a term that evaluates to true or false, depending on the value of x. $rule_x$ and $rule_0$ are rules. We arbitrarily pick a value x, from the universe U, that makes the term P_x true. We then execute $rule_x$ with that value of x. Typically, the variable x appears somewhere in $rule_x$, so what the rule does depends critically on what value for x is chosen. If no such value x exists, then we execute $rule_0$ instead. x usually does *not* appear in $rule_0$. In many cases, we can omit the **ifnone** clause — if we're sure there's always a value to choose for x, or if there's simply nothing to do in the case where no x exists. Also, if the choice of x is to be made over the entire universe U, we can omit the P_x term.[15]
Q: So the program above says to pick the proper position of the needle in the haystack, verify that the needle is in fact there, and return the position where we found it.
A: Correct.
Q: Again, I have this nagging feeling that we've cheated somehow. We seem to be magically "guessing" the correct answer and then verifying that our guess is correct.
A: Sure. But this serves well as a specification of the problem, doesn't it?
Q: Yes. Still, there's something about this high-level ASM that's different from the others we've seen. It's not just that the ASM "magically" finds the needle. Let's say there are multiple instances of the needle in the haystack. It seems like the ASM just randomly chooses one of them to return.
A: This is a non-deterministic algorithm, as opposed to the deterministic algorithms we've considered up to this point.
Q: So the ASM could be run twice on exactly the same inputs and return different results?
A: That's right.
Q: Why would this be a useful feature for ASMs? Computers are deterministic.
A: As in other algorithms, it is often useful to be able to talk about non-deterministic algorithms, since they may be more natural than their deterministic counterparts, and since we know that any non-deterministic algorithm can be translated into a deterministic one.
Q: Which may require exponentially more time to run.
A: True. But we don't always have to pay an exponential speedup cost, and we may or may not be interested in the running time of the algorithm.[16]

Q: This **choose** construct sounds awfully powerful — almost too powerful.
A: Certainly; one has to evaluate the reasonableness of the test predicate used to make the choice.
Q: Fair enough. Let's look at the "practical" implementations of this algorithm.
A: In general terms, what does a "practical" implementation do?
Q: In the haystack, there are a number of candidate substrings that could be matches for the needle. A matching algorithm must systematically try each candidate, one at a time. If and when a match is found, the algorithm terminates and reports it. If none turns out to be an actual match, the algorithm terminates with a negative answer.
A: Let me express what you just described in ASM terms:

if not finished **then**
    if noMoreCandidates **then** *Report failure*
    elseif candidatePossibleMatch **then** *Test candidate further*
    else *Try next candidate*
    endif
endif

This isn't a full-fledged ASM — it's a template, where the terms and rules are really names of *macros.* A macro consists of a name and a *replacement.* To give a complete ASM, we must give a replacement for each macro name. The replacements for finished, noMoreCandidates, and candidatePossible Match will be ASM terms, and those for *Report failure*, *Test candidate further*, and *Try next candidate* will be ASM rules.
Q: So, we start with the template ASM, replace each macro name with the replacement, and we get a complete ASM.
A: Exactly. Let's start by defining macro replacements to represent the brute-force approach.
Q: OK. In the brute-force approach, we work left to right, testing every potential starting index in the haystack. So, we'll write rules that simulate looping through all these indices.
A: Let's try this. We'll define a nullary function Index that holds the current index that we're testing. We'll assert that its initial value is 0. If we determine that the current value of Index is not the start of a match, we simply increment Index:

rule *Try next candidate*: Index := Index + 1

Q: But eventually we'll run out of possible matches.
A: Right. If we find no matches anywhere in the string, we update Index to undef:

rule *Report failure*: Index := undef

When can we stop trying values for Index and simply update it to undef?
Q: We don't have to go all the way to the end of the string. When we get to index Length(Haystack) - Length(Needle), we know there are not enough characters left in the haystack to give a match.
A: So if Index is greater than Length(Haystack) - Length(Needle), we know there is no match in the haystack:

term noMoreCandidates: Index > Length(Haystack) - Length(Needle)

Q: OK. But there are more macro replacements to define. How about the term candidatePossibleMatch?
A: Given our Substr function, we can determine immediately whether a candidate is a match. Just check whether the needle-length substring of the haystack starting at Index is identical to the needle:

term candidatePossibleMatch: Substr(Haystack,Index,Length(Needle)) = Needle

Q: If this term evaluates to true, the candidate isn't just a *possible* match — it *is* a match.

A: Right. The Substr function gives us the power to test a candidate in a single step. Therefore, the rule *Test candidate further* simply records the successful match:

rule *Test candidate further*: Output := Index

Q: There's one more macro replacement to define — the term finished.
A: Can you define it?
Q: The algorithm is finished if a match is found — in which case Output is updated to some value — or if no match is found anywhere — in which case Index is updated to undef. So we could define finished as follows:

term finished: Output $\neq$ undef or Index = undef

A: That's correct.
Q: Could I see the whole ASM, with the macro replacements pasted into the proper places in the template?
A: Sure. Here it is.[17]

```
if Output = undef and Index ≠ undef then
    if Index > Length(Haystack) - Length(Needle) then Index := undef
    elseif Substr(Haystack,Index,Length(Needle)) = Needle then Output := Index
    else Index := Index + 1
    endif
endif
```

Q: This program appears to be linear-time, but I was always taught that the naïve algorithm was quadratic in time.
A: It is linear time if you consider the process of comparing two strings for equality to be a constant time operation. Of course, this operation takes more time with longer needles.
Q: How could we reflect that?
A: We could produce yet another set of rules that compare the needle against the haystack character by character. Let's define another nullary function Offset, which will be used in the character-by-character comparison. The needle character at index Offset will be compared with the haystack character at index Index + Offset. A candidate is successful — at least, for the time being — if the needle and haystack characters match:

term candidatePossibleMatch: CharAt(Needle, Offset) = CharAt(Haystack, Index+Offset)

Q: Here you've replaced the string equality test and the function Substr with a *character* equality test.
A: Right. If you think about it, we did a similar thing in the gcd example; we replaced the high-level function gcd with lower-level rules to compute the values for this function.
Q: I notice that you've sneaked in a new function CharAt. It appears that CharAt takes a string and an integer and returns the character at that position in the string.
A: Exactly. See how useful good names can be?
Q: Indeed. And it looks like we did need the Character universe after all.
A: You're right; we did — but only now that we're at the low abstraction level of character-by-character comparison. Our first three ASMs didn't require such a universe.
Q: Now, we're not necessarily done just because we find a single character match in the needle and haystack.
A: Right. We need to increment Offset — unless the number of character matches we've found equals the length of the needle, in which case we can report success:

```
rule Test candidate further: if Offset = Length(Needle) then Output := Index
                             else Offset := Offset + 1
                             endif
```

Q: And if we find a character *mis*match?
A: As before, we increment Index, but we must also reset Offset:

rule *Try next candidate*: Index := Index + 1
Offset := 0

Q: I was going to ask about the definitions of the remaining macros, but now that I think about it, it looks like they can remain unchanged from their last definitions.
A: Right. That's an advantage of breaking the ASM up into components. Of course, macros are a rather primitive way of defining components, but they seem to work well here.
Q: I hate to put you through it again, but could you plug these new macro replacements into our template?
A: No problem.

```
if Output = undef or Index ≠ undef then
    if Index > Length(Haystack) - Length(Needle) then Index := undef
    elseif CharAt(Needle, Offset) = CharAt(Haystack, Index+Offset) then
        if Offset = Length(Needle) then Output := Index
        else Offset := Offset + 1
        endif
    else
        Index := Index + 1
        Offset := 0
    endif
endif
```

Q: This algorithm certainly seems to reflect the quadratic nature of the naïve algorithm; I can fairly easily see the nested loop of iterations. If memory serves, the Knuth-Morris-Pratt algorithm [30] is a linear-time improvement on this algorithm.
A: How does Knuth-Morris-Pratt improve on the naïve algorithm?
Q: With the naïve algorithm, we may find ourselves backing up and re-examining characters; that's why it runs in quadratic time. Suppose we've matched n characters of the needle against the haystack when the next character fails to match. With the naïve algorithm, when we reset Offset to 0 and Index to Index+1, we end up re-examining characters Index+1 through Index+n of the haystack. But having seen those characters already, we shouldn't have to look at them once again. Knuth-Morris-Pratt adds a rule that allows us to avoid re-examining characters in this case.
A: Give me an example.
Q: OK. Say our needle is "BAAABB" and our haystack is "BAAABAAABB". With Index = 0, we get all the way to Offset = 5 before we get a mismatch. The naïve algorithm would increment Index to 1 and reset Offset to 0. But there's no need to re-examine characters 1 through 3 in the haystack, since they are all "A" and can't possibly be the beginning of a match. Character 4 in the haystack is "B", which could be the beginning of a match — in fact, it *is* the beginning of a match, as it turns out. So, we may as well skip ahead in the haystack by four — in other words, increment Index by 4. Of course, we also need to *back up* in the needle by four, so that we keep examining the same position in the haystack, and try matching from there.
A: That makes sense. So how do we take advantage of this knowledge?
Q: By means of a pre-computed table. When a mismatch occurs, the table tells us how far to back up in the needle without changing position in the haystack.
A: What information do we need to index into this table?
Q: Just the value of Offset.
A: OK. How about if we introduce a function Skip: Nat → Nat?
Q: That looks the right idea. It appears that Skip is the pre-computed function that tells us how far to back up in the needle, given the current value of Offset.

A: Correct. Looking at our earlier example, when Offset = 5, we saw that Index should be incremented by 4 and Offset should be decremented by 4. So what does that tell us about Skip?
Q: Skip(5) = 4.
A: Right.
Q: But how are the values for Skip computed?
A: We can just assume that they are fixed in the initial state. Our ASM deals only with the string matching itself, not the preprocessing that computes the table values.
Q: OK. By the way, there is one thing that I forgot to mention. It's possible that when we find a mismatch, there's no value in retesting *any* of the previous needle positions against the haystack, and we should skip over *all* the previously matched characters, to the next position in the haystack. In that situation, our pre-computed table returns a special value that tells us to move on.

For instance, if our needle is "AAAA" and our haystack is "AAABAAAA", with Index = 0, we get to Offset = 3 before we get a mismatch. Because character 4 in the haystack is not "A", none of the characters we've examined so far can be the beginning of a match. We may as well increment Index by 4.
A: OK. Let's say that Skip(Offset) returns Offset+1 if this happens.
Q: But we don't want Offset to be decremented by that much — that would make Offset negative.
A: Right — we'll make a special case for that. So then the rule macro *Try next candidate* will look like this:

rule *Try next candidate*: Index := Index + Skip(Offset)
if Offset > Skip(Offset) **then** Offset := Offset - Skip(Offset)
else Offset := 0
endif

Q: That seems right. But, of course, we've assumed the existence of the Skip function. In the "real" algorithm, we need to compute Skip before we proceed with the actual pattern matching.
A: We can certainly fix that. On the other hand, the program above could be viewed as complete, just at a high level of abstraction.
Q: I guess that's true. We could always extend the ASM to compute Skip.[18]
A: Right. Now, you mentioned the Boyer-Moore algorithm [12] earlier. How is Boyer-Moore different?
Q: In Boyer-Moore, the needle is compared against the haystack from right to left. The needle still advances along the haystack in a left to right fashion.
A: So, the program should be similar to that above, except that we decrement Offset instead of incrementing it. We'll have to change *Test candidate further*:

rule *Test candidate further*: **if** Offset = 0 **then** Output := Index
else Offset := Offset - 1
endif

Q: There's a bit more to it. Boyer-Moore uses the same rule as Knuth-Morris-Pratt to skip over certain characters. But it also uses another rule. As an extreme example, if the character currently being scanned in the haystack doesn't appear *at all* in the needle, we know that nothing to the left of this current character can be part of a match. So we can move the needle all the way to the right of the current character. Sometimes this new rule gives us a bigger skip; sometimes the old Knuth-Morris-Pratt rule does. Obviously, we skip by the maximum of the two.[19]
A: It sounds like this rule uses different information from the other. It's not the offset value that counts; rather, it's the current character in the haystack.
Q: Right. Maybe we need an additional Skip function — one that maps *characters* to natural numbers.
A: Let's try this. Skip1 and Skip2 will be functions that give the skip values of the first and second rules, respectively. *Try next candidate* will look like this:

rule *Try next candidate*: **if** Skip1(Offset) > Skip2(CharAt(Haystack, Index + Offset)) **then**
    Index := Index + Skip1(Offset)
  else Index := Index + Skip2(CharAt(Haystack, Index + Offset))
  endif
  Offset := Length(Needle) - 1

Q: You're assuming that Offset has the initial value of Length(Needle) - 1, right?
A: Correct — just as we implicitly assumed that Offset had the value 0 in the previous program. I suppose we really should be more careful about stating those conditions.
Q: I guess we're going to "skip" the details of Skip1 and Skip2. But let me ask you: could we combine Skip1 and Skip2?
A: Yes, we could. We could define a function Skip: Nat $\times$ Character $\rightarrow$ Nat that gives us the maximum of the results from both rules. Then the conditional rule that updates Index could be replaced with a simple update:

Index := Index + Skip(Offset, CharAt(Haystack, Index + Offset))

Minimum Spanning Tree

Q: We have one more example to consider: the minimum spanning tree problem.
A: Right. Where should we begin? You mentioned earlier that you had a few different algorithms in mind for solving this problem.
Q: Well, we keep talking about abstraction as a design principle, so I suppose we could start with trying to find a single abstract algorithm that captures the various algorithms.
A: You sound a little hesitant.
Q: At this point, I'm not sure that I see the relationship between the algorithms I have in mind.
A: Well, let's start by looking at the detailed algorithms first; perhaps in comparing them, we can find the commonalities and create the abstract version later.
Q: Isn't that doing it backwards?
A: I suppose it doesn't really matter what order you use if you get to the same results. That's an important point to recognize as you work on writing ASMs. Usually, when we present our ASMs to someone else, we start with the most abstract and move towards the most concrete. That doesn't mean that they were written in that order.
Q: Ok. Let's start with Prim's algorithm [27, 32].
A: Describe it for me.
Q: It's a greedy algorithm. We construct a tree from an arbitrary node by repeatedly selecting the minimum-weight edge that extends the tree to a node not currently in the tree.
A: We'll keep track of the edges we've selected by means of a function Tree: Edge $\rightarrow$ Boolean. We'll also track the points in the graph that have been selected by means of a function Selected: Node $\rightarrow$ Boolean.
Q: So, Tree(e) will be true if e has been selected to be an edge in the tree, and false otherwise.
A: Correct. Similarly, we'll use the function Frontier: Edge $\rightarrow$ Boolean to indicate edges that are candidates for being selected as the next edge of the tree.
Q: Why the name "frontier"?
A: As the algorithm progresses, two sets of points emerge: those already in the minimum spanning tree and those not yet in the tree. The frontier is the set of edges that connect points in the tree with those not in the tree; as the tree "advances" through the graph, an edge from the frontier is always selected.
Q: I see. Is that all we need for the program?
A: We'll be using another ASM rule form in this program as well. Let's begin by looking at part of the program for Prim's algorithm, namely, how we initialize the program:

```
if CurrentMode = initial then
    choose p: Node
        Selected(p) := true
        do-forall e: Edge: p ∈ Endpoints(e)
            Frontier(e) := true
        enddo
    endchoose
    CurrentMode := run
endif
```

Q: What's this "do-forall" rule?
A: The general form of this rule is

```
do-forall x: U: P_x
    rule_x
enddo
```

As you can see, it looks quite similar to the **choose** rule. x, U, P_x, and $rule_x$ are all defined similarly to the **choose** rule. To execute the **do-forall** rule, do the following: for each value of x that makes $\mathsf{P}_x = \mathsf{true}$, execute $rule_x$ with that value for x. These executions of $rule_x$ occur in parallel.
Q: This seems even more powerful than the **choose** rule. You can do an infinite amount of work with this rule.
A: True; as with **choose**, one needs to be careful about how much work is performed.[20] Notice that in this example that the universe Edge over which the variable e ranges is a finite universe, so only a finite number of Edges are selected for processing.
Q: Then why not just create rules that walk through the set of edges one by one and do the required processing? Most programs would have to take that approach.
A: Notice that in that approach, you have to place an order on the edges in the set. This order is meaningless; does it really matter which edge is the fifth to be processed?
Q: I suppose not. And the presence of an ordered list might imply to a reader that there is some significance to the ordering. Is that what you're getting at?
A: Exactly. Getting back to the algorithm itself, do you understand what the rule above does?
Q: It appears that it selects the arbitrary starting point for the tree and marks all the adjacent edges as frontier edges.
A: Correct. Now, we move on to the rest of the program:

```
if CurrentMode = run then
    choose e: Edge: Frontier(e) and ((∀f: Edge) Frontier(f) ⇒ Weight(f) ≥ Weight(e))
        Tree(e) := true
        choose p: Node: p ∈ Endpoints(e) and not Selected(p)
            Selected(p) := true
            do-forall f: Edge: p ∈ Endpoints(f)
                if Frontier(f)=true then Frontier(f) := false
                else Frontier(f) := true
                endif
            enddo
        endchoose
    ifnone CurrentMode := done
    endchoose
endif
```

Q: Let me see if I can understand this rule. We choose a frontier Edge of minimum Weight and mark it as a tree Edge. I notice that you use a universal quantifier to make that selection.
A: True; we should have mentioned that earlier. Terms may include universal or existential quantifiers. Of course, one may be concerned about the amount of computation implied by such terms. There may be a lot of computation hidden inside a quantified term. How much work is it to verify that a frontier edge has minimal weight? Possibly a lot, depending on the implementation and the size of the graph. But once again, since the Edge universe is finite, there is only bounded work involved here.
Q: Indeed. What does the inner **choose** rule do?
A: First, we pick a point that is an endpoint of the new tree edge but isn't already selected as a tree point. How many points are there?
Q: Well, each edge has two points, which we've been implicitly assuming are different, right?
A: Right.
Q: And since this edge is a frontier edge, one of its endpoints is selected and one is unselected, so there is exactly one point to be processed by the rule.
A: Exactly right. That point is the one added to the tree.
Q: There's something slightly misleading about the use of **choose** here. In general, the choice involved in a **choose** rule is nondeterministic, but here it's deterministic.
A: You have a point. It would be a good idea to add a comment to that effect in your ASM. Here's a case where natural language can play a useful role in documentation.
Q: OK. Now, what does the **do-forall** rule do?
A: Consider all the edges that touch the newly selected point p. Those which were previously in the frontier had p as the "outside" point; since p is now "inside" the tree, those edges cannot be used to form the tree and should be removed from the frontier. On the other hand, those edges which weren't in the frontier and touch p now meet the definition of frontier edge and should be added to the frontier.
Q: We could replace that innermost conditional rule with a simple update rule:

$\mathsf{Frontier}(f) := \mathsf{not\ Frontier}(f)$

A: Yes, you're right.
Q: It occurs to me that you've been thinking about the frontier as a set, even though it's formally defined as a Boolean-valued unary function.
A: Indeed; as you may recall from discrete mathematics, a set can be described as a Boolean-valued function, called the "characteristic function" of the set. Formally speaking, all of the universes we've seen using are really Boolean-valued functions.
Q: So how do you distinguish between unary Boolean-valued functions that are universes, like Edge, and unary Boolean-valued functions that are *not* universes, like Frontier?
A: Really, there's no important difference between them. In this example, you can take the set of "frontier points" to be a subuniverse of Node.
Q: OK. So, are we done?
A: Sure; let's move on. You mentioned another algorithm for solving the minimum spanning tree problem. Which one is that?
Q: Kruskal's algorithm [31]. The idea is to select the minimum weight edges available anywhere in the graph. In general, this means that during the algorithm, there may be several small connected components that have been selected. In each stage, we select the cheapest eligible edge until we have a complete tree.
A: How do you know whether an edge is eligible for selection?
Q: We have to check that adding an edge doesn't form a cycle. In Kruskal's algorithm, the nodes of each connected component are labeled with a unique common identifier. Typically, the label is a reference to one of the points in the component. We select the minimum weight edge that does not connect two points with the same identifier. Additionally, since adding an edge connects two previously disconnected components, we must re-label the points in one of the components so that all points in the new supercomponent have the same label.

A: All right; that's probably enough detail to proceed. Rather than define all the functions carefully, let's proceed immediately to the program:

```
if CurrentMode = initial then
    do-forall p: Node
        Label(p) := p
    enddo
    CurrentMode := run
elseif CurrentMode = run then
    choose e: Edge: Eligible(e) and ((∀f: Edge) Eligible(f) ⇒ Weight(f) ≥ Weight(e))
        Tree(e) := true
        choose p, q: Node: {p, q} = Endpoints(e)
            do-forall r: Node: Label(r) = Label(p)
                Label(r) := Label(q)
            enddo
        endchoose
    ifnone CurrentMode := done
    endchoose
endif
```

Q: The condition in the **choose** rule looks similar to the one from Prim's algorithm — it's just ensuring that the chosen edge has minimal weight. But it uses a function Eligible, and I don't see it updated anywhere.
A: Good point. Actually, Eligible is a name of a macro:

```
term Eligible(e):
(∀p, q: Node) {p, q} = Endpoints(e) ⇒ Label(p) ≠ Label(q))
```

It simply checks whether the nodes p and q that make up the endpoints of the given edge have different labels.
Q: OK. I see p and q again, in the inner **choose** rule. What does that rule do?
A: At this point, we've already selected our Edge e to be considered for inclusion in the tree. We need to give different names p and q to the two endpoints of that Edge; this is a non-deterministic choice.
Q: Although the correctness of the algorithm doesn't depend on which of the two points is assigned to p or q.
A: True enough. But the running time of the algorithm may very well depend on that choice. The choice of p and q is significant in that all the nodes labeled with p are relabeled with q. So it's best from a performance standpoint if the q labels outnumber the p labels. If we continually make the "wrong" choice for p and q, we may end up re-labeling a lot of points over and over again.
Q: I see; that is probably important enough information to preserve. By the way, that inner **choose** rule chooses values for *two* variables. Is that OK?
A: Good observation. Strictly speaking, we should really use a **choose** rule for q nested inside a **choose** rule for p. But the single **choose** rule is clear and concise.[21]
Q: OK. That seems reasonable.
A: Now, let's compare the ASM for Prim's algorithm with the one for Kruskal's algorithm. Do you see any similarities?
Q: At this level, I can see some similarities. At each step, we select an Edge with minimum Weight out of a set of eligible Edges, and re-process the set of eligible Edges to make the next selection.
A: Indeed. We could produce a template for greedy minimum spanning tree algorithms, which would express the commonalities between such algorithms.
Q: You're not going to make me do it now, are you?
A: No. Perhaps you can try it on your own.[22]

Verification

Q: OK. I can see how ASMs can be used in design. Now can we talk a bit about verification?
A: This is one of the fundamental problems of software engineering. How do you ensure that what you've built meets the specification?
Q: There are several techniques: code reviews, various forms of testing, and formal proofs of correctness.
A: And all of these can be used with ASMs. Let's start with code reviews. What's the purpose of a code review?
Q: It's intended to uncover defects in the code, through inspection by a wide variety of experts. I suppose that an ASM can be considered "code", can't it?
A: Certainly. We can perform code reviews on ASMs — in fact, we've been conducting reviews even as we were constructing the rules above.
Q: True. Is there anything about ASMs that makes them particularly suited to code reviews?
A: Yes. Any formal description technique such as ASMs encounters the problem of the "ground model" [8]. It's certainly possible to verify whether one formal model is equivalent to another formal model. But how can you verify whether a formal model corresponds to the "real-world" system that it's supposed to represent?
Q: I suppose the best that one can do is to examine the model and convince yourself that it is a faithful description of the real-world system.
A: Indeed. An advantage of using ASMs is that we can describe the system using the terms and concepts that are natural to the system. In principle, that makes the process of verifying the ground model easier, since we don't have to figure out how those concepts are encoded in the formal framework.
Q: So, domain experts don't have to be expert programmers to examine an ASM and offer their opinions on its correctness.
A: That's exactly the point. ASMs attempt to keep the notational overhead at a bare minimum in order to make that analysis easier.
Q: How about testing?
A: ASMs are executable. There are a variety of tools available for such purposes; the most recently developed tools are AsmL [4] and XASM [38]. You can test your ASM by executing it directly.
Q: It occurs to me that with ASMs, you can test at almost any point in the development process — not just at the end when you have a full implementation.
A: Certainly — and with added benefits, since errors detected at earlier points in the development process are cheaper to fix, as you well know. All the usual benefits of testing apply to ASMs as well.
Q: How about formal *proofs* of correctness? I remember hearing about them in my software engineering class. Since ASMs have a formal basis, I suppose that they are really intended for this kind of verification.
A: I wouldn't say that ASMs are intended for any *particular* kind of verification. Many formal methods practitioners promote verification as the true test of the usefulness of a formal method. But that's an awfully limited view of ASMs.
Q: I suppose that makes sense. Even in our discussions today, I've learned a lot about the algorithms we've discussed as we've worked on expressing them as ASMs.
A: And those insights are extremely valuable. In fact, a proof is yet another way of expressing those insights to an outside audience. But it doesn't have to be the only way of expressing those insights.
Q: I don't usually think of proofs as a method of communication. I tend to think of lots of Greek letters and confusing definitions.
A: And you wouldn't be alone in that regard. But you're thinking of an extremely formal notion of proof. A proof is essentially a really convincing argument — nothing more complicated than that. The level of formality is up to you. If you're trying to convince a colleague that the software module you just designed works the way it should, you give him or her a little proof — it will most likely be given in an informal, conversational style, but it's still a proof.
Q: That seems like a much less intimidating definition of proof.
A: Yes. Let's put it into practice. Shall we try proving a property of one of our ASMs?

Q: OK. Which ASM?
A: Let's use the one we just constructed — the ASM for Kruskal's algorithm.
Q: And what should we prove about it?
A: Well, the algorithm terminates with a set of tree edges chosen. Assuming that the graph is connected — that there's a path between any two nodes — this set of edges should form a spanning tree. Furthermore, this spanning tree should have the lowest weight of all possible spanning trees. Let's prove the first property — that the algorithm chooses a spanning tree.
Q: You'll have to help me here — I'm not sure how to begin.
A: This is often the hardest part of a proof — getting the insight into why the algorithm works. Let's consider what happens when the algorithm terminates. Say there are n nodes in the graph. At termination, all n nodes have the same label, they are connected by tree edges, and there are exactly $n-1$ tree edges.
Q: Why $n-1$?
A: Any fewer than $n-1$ wouldn't be enough to connect the nodes, and any more would form a cycle.
Q: OK. But how does the algorithm get to this state?
A: At each state, there are various sets of nodes labeled with the same label. For any label p, let L_p be the set of nodes with label p, let n_p be the number of nodes in L_p, and let t_p be the number of tree edges adjacent to nodes in L_p.[23] We claim that at every state in a run of the ASM, the nodes in any nonempty L_p are connected by tree edges, and $t_p = n_p - 1$. In other words, there's a miniature spanning tree for each set of nodes L_p.
Q: Eventually, there is just one L_p that is nonempty, so L_p contains all the nodes in the graph. So your claim implies that at this point we have a spanning tree chosen for the entire graph.
A: Right. Now, let's prove the claim by induction. Our induction variable will be the number of steps taken so far by the ASM. Let's start just after the first step, in which all nodes' labels are initialized. Immediately after this step, does the claim hold?
Q: Each node is labelled with itself, so each L_p is a set containing just one node, p. I suppose the nodes in L_p are connected, in a trivial way. There are no tree edges, so the number of tree nodes adjacent to p is 0, which is $n_p - 1$. So the claim does hold.
A: OK. Now let's say the claim holds at some state and then the ASM takes a step. What happens?
Q: An edge e, whose endpoints have different labels p and q, is chosen as a tree edge. The nodes labeled with p are relabeled with q.
A: Right. So L_q grows in this one step. Let L'_q be the number of nodes labeled with q after the step. Likewise, let n'_q be the size of L'_q, and let t'_q be the number of tree edges adjacent to nodes in L'_q. How many nodes are in L'_q?
Q: The nodes in L_p plus the nodes in L_q. So, $n_p + n_q$.
A: How many tree edges are adjacent to nodes in L'_q?
Q: There are $n_p - 1$ edges from L_p and $n_q - 1$ edges from L_q. And there's the edge e between p and q. So that's $n_p + n_q - 1$.
A: And are all the nodes in L'_q connected by tree edges?
Q: The nodes in L_p are connected, the nodes in L_q are connected, and the edge e connects nodes in L_p to nodes in L_q. So, yes.
A: Good — so the claim holds for the label q. Note that after this step, L_p is empty. Moreover, for any label r other than p or q, L_r and t_r are unchanged, so by the inductive hypothesis, the claim holds for r. So, we've proved the claim.
Q: Is that it?
A: There's one more step. You need to show that the number of labels eventually reaches one. This isn't too hard — note that we start with n labels and remove one label per step.[24]
Q: Ok, I can understand the proof. What did this have to do with ASMs?
A: Well, ASMs allowed us to talk formally about the properties of an execution of the algorithm. Notice that we performed our induction over the number of steps taken by the ASM. This was a concept that we defined earlier and that we implicitly use all the time. Can you imagine doing this with an algorithm written in C or Java?

Q: It certainly would seem more difficult. So, ASMs provide you with some of the tools and terminology that it would take to write a proof.
A: Exactly. We could have proved these properties any number of ways; we just picked the techniques that seemed the most natural to the problem.
Q: How common are proofs of correctness using ASMs?
A: It depends on the application area. In some cases, just getting a precise and understandable ground model is enough. In others, part of the motivation for a formal description is the ability to prove correctness. By now, there are so many examples of ASM-based proofs that it would be hard to list them all.
Q: Are all those proofs crafted by hand? I seem to recall hearing about automated tools for verification.
A: Most are hand-crafted, in part because the intended audience for most proofs is still humans, not machines. Still, there is substantial work being done on supporting mechanical verification, using theorem provers or model checkers [34, 39, 15, 17, 37].

Final Thoughts

Q: We seem to have reached the end of the waterfall. I suppose that I should start trying these ideas out by constructing a few ASMs.
A: Indeed. One often learns best by doing.
Q: I'd still like to see ASMs applied to a real-world example.
A: Yes, the examples we covered were rather simple. We can get together again and look at something more substantial.
Q: I'd like that. In the meantime, what else do you suggest?
A: There are over two hundred papers that have been written using ASMs applied to various domains [1]. If you have a particular background in one of those domains, you might try picking up a paper and see how the concepts and ideas you're most familiar with are expressed in the ASM world.
Q: Can you give me a taste of what else is out there?
A: Certainly. The ASM method has been applied to a number of domains outside of pure software development. Hardware systems, hybrid hardware-software systems, mathematical logic and complexity theory, and natural language systems are a few of those domains.
Q: The notion of "algorithm" that an ASM captures doesn't seem to be restricted to software. Anything that embodies an algorithm can be described in terms of an ASM, not just software artifacts.
A: Right. In addition, we explicitly considered the case of a single agent program running in discrete, sequential time; multi-agent extensions of the ASM method are available for those interested in distributed systems or real-time systems.
Q: Perhaps we can sit down some other time and discuss how to write a multi-agent ASM.
A: We'd look forward to it.

Acknowledgments

Numerous people have suggested the need for a practical guide to ASMs over the years; Uwe Glässer, Wolfgang Reisig, and Kirsten Winter are among those we remember making the suggestion. Many of the examples given above have been taken from various talks we have heard over the years. We are indebted to the numerous ASM authors who have written ASM introductions in their own papers.

The discussion of requirements and specifications uses ideas from Jackson [26] and Gunter *et al.* [18].

We are grateful to Yuri Gurevich of Microsoft Research for his comments and support; our student Questor is a friend of Yuri's long-time colleague Quisani[20, 22, 6].

Finally, we thank the students in the fall 2002 section of CS 4712 "Software Quality Assurance" at Michigan Technological University, the first users of this primer: John Atkinson, Anshu Bhatia, Brian Bos, Michael Burke, Richard Kuchera, Alok Mishra, Parul Mishra, Nicholas Negoshian, Jacob Northey, Nisha Oommen, Allen Tipper, Andrew Tomaszewski, and Derek VerLee.

Of course, any omissions or errors are our own. Corrections or comments from readers are most welcome.

References

[1] Abstract State Machines home page. `http://www.eecs.umich.edu/gasm/`.

[2] ASM Workbench home page.
`http://www.uni-paderborn.de/cs/asm/ASMToolPage/asm-workbench.html`.

[3] ASMGofer home page. `http://www.tydo.de/AsmGofer`.

[4] AsmL home page. `http://www.research.microsoft.com/foundations/asml`.

[5] K. Beck. *Extreme Programming explained.* Addison-Wesley, 2000.

[6] A. Blass and Y. Gurevich. New zero-one law and strong extension axioms. *Bulletin of EATCS*, 72:103–122, October 2000.

[7] B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[8] E. Börger. High level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.

[9] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *Journal of Universal Computer Science*, 8(1):2–74, 2002.

[10] E. Börger and U. Glässer. Modelling and analysis of distributed and reactive systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras – Mini-Course, BRICS Technical Report (BRICS-NS-95-4)*, pages 128–153. University of Aarhus, Denmark, July 1995.

[11] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.

[12] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[13] Boyer-Moore fast string searching algorithm: Home page.
`http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/index.html`.

[14] F. Brooks. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[15] G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proceedings of TACAS 2000*, number 1785 in LNCS, pages 331–346. Springer, 2000.

[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[17] A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Gurevich et al. [24], pages 303–322.

[18] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000.

[19] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, EECS Department, University of Michigan.

[20] Y. Gurevich. Evolving Algebras: A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.

[21] Y. Gurevich. Evolving Algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[22] Y. Gurevich. Sequential ASM thesis. *Bulletin of EATCS*, 71(67):93–124, February 1999.

[23] Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[24] Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines: Theory and Applications*. Number 1912 in LNCS. Springer, 2000.

[25] Y. Gurevich, N. Soparkar, and C. Wallace. Formalizing database recovery. *Journal of Universal Computer Science*, 3(4), 1997.

[26] M. Jackson. *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[27] V. Jarnick. O jistem problemu minimalnim. *Acta Societatis Scientiarum Natur. Moravicae*, 6:57–63, 1930.

[28] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[29] D.E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison Wesley, third edition, 1997.

[30] D.E. Knuth, J. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–360, 1977.

[31] J. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[32] R.C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, 36:1389–1401, 1957.

[33] W.W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proc. IEEE WESTCON*, pages 1–9, 1970.

[34] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[35] M. Sipser. *Introduction to the Theory of Computation*. PWS, 1997.

[36] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.

[37] K. Winter. Towards a methodology for model checking ASM: Lessons learned from the FLASH case study. In Gurevich et al. [24], pages 341–360.

[38] XASM home page. `http://www.xasm.org/`.

[39] W. Zimmerman and T. Gaul. On the construction of correct compiler back-ends: An ASM approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.

Notes and exercises

[1]**Note.** For rigorous definitions of ASMs, consult the "Lipari guide" [21] and the "1997 ASM guide" [19]. These and many other ASM resources are available at the ASM website [1]. By the way, ASMs used to be called "Evolving Algebras", so earlier ASM papers like the Lipari guide use this name.

[2]**Note.** This will have to be left to another conversation. We intend to discuss the database recovery problem covered in Gurevich *et al.* [25], which has a non-trivial specification.

[3]**Exercise.** Consider a slightly different problem: computing the greatest common divisor of two *natural numbers* (nonnegative integers). Modify the universe and function definitions in this section to fit this alternate problem.

[4]**Exercise.** Give another reasonable interpretation of Substr.

[5]**Exercise.** Does the presence of negative weights cause difficulties in the minimum spanning tree problem? Explain.

[6]**Exercise.** Give *two* sets of universe and function definitions to describe *directed* graphs: one that includes a universe of Edges and one that does not.

[7]**Note.** Actually, there *is* one rule simpler than an update rule. The *skip rule* is represented by the keyword **skip**. Executing this rule results in *no updates*. This is useful in certain circumstances, but it is not often used.

[8]**Note.** With simultaneous updates comes a slight complication: what happens when different updates to the same location are executed? For instance, what value does flag have after the following rule is executed?

```
flag := true
flag := false
```

The official answer is that if any updates conflict in this way, the ASM simply stops making updates; it "halts". One *could* exploit this to make the ASM halt whenever desired; simply issue conflicting updates when it's time to halt. But it's certainly not a reader-friendly technique; it's much clearer to issue an update like CurrentMode := done. Simultaneous, conflicting updates should be avoided.

[9]**Exercise.** Give an ASM that swaps the values in nullary functions A and B by temporarily storing A's old value in a nullary function temp, copying B's old value to A, then copying temp's value to B. How many steps does the ASM take to perform the swap?

[10]**Note.** The definitions of run, program, and state presented here are slightly simplified versions of those in the Lipari guide [21].

[11]**Exercise.** Earlier, we asserted that A and B return PosInts and that mod only takes PosInts as arguments. But notice that in our ASM, A and B are updated; this raises the question of whether either A or B can ever be updated to 0. Argue that if A and B are initially PosInts, they are never updated to 0 (and therefore, mod never takes 0 as an argument).

[12]**Exercise.** Modify the ASM so that it computes the gcd of any two *natural numbers* A and B.

[13]**Exercise.** Euclid's algorithm is an elegant solution to the gcd problem, but there are other possibilities. Consider the following "brute force" solution. Let m be min(A,B) initially. Look for values n and p such that $m \cdot n = \mathsf{A}$ and $m \cdot p = \mathsf{B}$. If such n and p are found, m is the gcd; otherwise, decrement m by 1. Note

that if m reaches 1, we're sure to find values for n and p: namely, $n = \mathsf{A}$ and $p = \mathsf{B}$. Construct an ASM to represent this algorithm.

[14]**Exercise.** Modify the ASM so that it computes the gcd without using either mod or CurrentMode.

[15]**Exercise.** Let U be a universe, and let p: U → Boolean and q: U → Boolean be functions. Let *R1* and *R2* be the following ASM rules:

rule *R1*: **choose** x: U: p(x)
    q(x) := true
  endchoose

rule *R2*: **choose** x: U
    if p(x) **then** q(x) := true
    endif
  endchoose

Consider the results of executing *R1* at a particular state S. Then consider the results of executing *R2* at S. Why might the results be different?

[16]**Exercise.** Say we've defined a function dividesEvenly: PosInt × PosInt → Boolean such that dividesEvenly(x, y) = true if and only if x divides y evenly. Give a **choose** rule that uses the dividesEvenly function to choose the gcd d of two PosInt inputs A and B in a *single step* and store it in a nullary function Output. Is the choice of g deterministic? Is an **ifnone** clause necessary?

[17]**Note.** The careful reader will note that the term in the main **if** rule should really be not (Output ≠ undef or Index = undef). It has been changed slightly, to Output = undef and Index ≠ undef, in order to avoid the confusion of a double negative.

[18]**Exercise.** Augment the ASM for KMP so that it computes Skip. You can take either a low-level or a high-level approach here. The low-level approach is more involved; consult some detailed description of the algorithm for guidelines [30, 16]. On a higher level, you can use the **choose** rule to choose an appropriate Skip value for each index. For each k, Skip(k) should be the least $j > 0$ such that the first $(k - j)$ characters of Needle match the $(k - j)$ characters of Needle starting at index j. If no such j exists, Skip(k) should be $k + 1$.

[19]**Note.** There is an HTML-based illustration of the Boyer-Moore and Knuth-Morris-Pratt algorithms in action, created by Moore himself [13].

[20]**Exercise.** Given a function f: Nat → Nat and a function g: Nat → Nat, give an ASM that copies all the values of f to g in a single step. Why is it *not* possible to write the following?

g := f

[21]**Exercise.** Rewrite the **choose** rule as a **choose** rule for q nested inside a **choose** rule for p.

[22]**Exercise.** Construct an ASM template, with appropriate macro names, that includes the basic components of any greedy minimum spanning tree algorithm. Give corresponding macro replacements for Prim's algorithm and Kruskal's algorithm.

[23]**Exercise.** Give formal definitions for L_p and t_p. Your definitions should be of the form

$$L_p = \{q : \phi(q)\}$$

$$t_p = |\{e : \psi(e)\}|$$

The task here is to find appropriate terms for $\phi(q)$ and $\psi(e)$.

[24]**Exercise.** Assume that the ASM is executed with a connected graph of n nodes. Prove that for any positive $k \leq n$, the number of labels still in use after k steps is $n - k + 1$. What happens at step $n + 1$?