Spectur — A Specification Language for the Programmer

Michael W. Godfrey
Richard C. Holt

September 19, 1994

Abstract

Spectur (*Spec*ification language after *Tur*ing) has been designed as a language for formal specification. The goals of the language include: ease of use by a specifier to write a software system specification from an informal description; ease of use by an implementor to build a program from a specification; ease of understanding by a user as documentation; ease of use by a mathematician to prove program correctness. The Spectur method of specifying a program module consists of declaring abstracted data structures, and specifying the procedures and functions of the module in terms of pre- and post-conditions on those data structures. Spectur is notable for providing notation and data structures that are both mathematically precise and immediately recognizable by programmers. A non-trivial example specification of a small operating system has been done using Spectur. We discuss this example, as well as the usefulness and potential applications of Spectur.

1 Formal Specification and Spectur

The importance of specification is understood well by the software engineering community. Specifications are useful both at the design stage of software development, to help to clarify and to decompose the problem, and later as documentation and as an aid in proving program correctness. A central aim of the Spectur (*Spec*ification language after *Tur*ing) project was to design a language that would be convenient for formally specifying large software systems. It was felt that while any such specification should ideally be both mechanically verifiable and mathematically well-defined, it should also be easy to understand. In particular, it was felt that the resemblance between the source code and the formal specification should be strong. In this way, not only would it be easy for an implementor to write a program from a specification, it would also be easy for a specifier engaged in reverse engineering to write a specification from an implementation. While there currently exist many different languages and approaches to formal specification (such as VDM [Jones 90], Z [Spivey 88] and Larch [Guttag et al. 85]), few of them concentrate on the issue of matching the structure and "feel" of the actual program. Spectur is notable for being immediately accessible and understandable to implementors.

2 Spectur: Goals and Realization

2.1 Goals of Spectur

The creation of Spectur was undertaken with a number of goals in mind. The basic motivation was to design a language for program specification that, while amenable to program correctness methods, is both expressive and easily readable by the implementor. For these reasons, it was felt that the overall structure of a specification should be similar to that of its implementation.

As to the actual language, it was felt that Spectur should be easy to learn, powerful, descriptive, mechanically checkable, semantically clean and promote the basic tenets of good software engineering. In particular, since Spectur specifications are to be usable in formal correctness arguments, the language must be mathematically definable. Also, since Spectur is to be used in the program development process, it should be easy to build software tools to support it. Finally, it was decided that Spectur should restrict itself to ASCII characters; in this way, specifications may be written at a normal keyboard.

2.2 Basic Approach

There are three well-known approaches to formal specification: operational, algebraic (also known as "axiomatic") and state-based (also known as "pre-post"). An operational specification of a program consists of another program written in a language with well-defined semantics, such as pure Lisp, Turing [Holt et al. 88] or the axiomatized subset of Pascal [Hoare and Wirth 73]. Although the specification program can in principle be considered to be an implementation, the primary design goals of the specifier are descriptiveness and clarity instead of efficiency. The traditional complaint against the operational approach is that it is overconstraining and blurs the distinction between a program's functionality and its implementation. It overspecifies the problem being considered by providing an explicit definition not only of the program's data structures, but also of the mechanics of all operations on them (rather than a simple description of the effects).

The algebraic approach takes a much different tack. Usually very mathematical in style, most of the effort with this method comes in determining an appropriate set of abstract data structures together with sets of operations on them. The operations are defined implicitly by listing axioms that describe their effects. The actual program specification is then often very brief. The major advantage of algebraic specifications is that they are well-suited to formal reasoning, since all operations are specified in sentences of mathematical logic. Unfortunately, they can also appear quite cryptic, and can be difficult to write without considerable practice. Algebraic specifications often bear little structural resemblance to the programs they specify.

The state-based approach considers a program to be a kind of large data structure whose "state" can be altered (or examined) by means of procedure calls by a user. The behaviour of the program is specified by first defining the initial state of the program and then describing explicitly how each procedure can modify the program state. A natural way to do this is via predicate logic pre- and post-conditions, which specify what must be true of the program state immediately before and after each procedure call.[1]

[1]The functional or "applicative" approach to formal specification is a special case of the state-based approach. However, an applicative specification has no internal state *per se*; rather, the state is passed as a parameter to all procedures, with a new state being returned. Thus the pre- and post-conditions of a procedure specification involve only the procedure's formal parameters.

The state-based approach seemed the most appropriate for Spectur for several reasons: The state-based paradigm conforms closely to the commonly held "mental model" of a program as a black box that accepts input in one slot and then produces output out of another — the emphasis is on what effects a procedure call might have on the program state, rather than on what mechanisms might be used to achieve them. Also, while algebraic specifications may be very convenient for use in correctness proofs, in practice it has been found that they can be very difficult to create and to understand. Since the primary design criterion of Spectur was ease of use and understanding, it was decided that the state-based approach was the most suitable for Spectur.

2.3 Data Structures

The description of the initial program state in the state-based approach usually involves the specification of some sort of abstract data structures. With some specification languages, such as Z [Spivey 88], the data structures have a very mathematical style — the rationale being that since mathematical objects such as sets and mappings are inherently abstract and subject to rigorous methods, they are appropriate for abstract problems and easy to reason about. However, it was decided that since Spectur specifications should closely resemble their implementations, Spectur should also provide data structures that have a definite programming language orientation. Thus in addition to numbers, sets and mappings, Spectur provides records, arrays, pointers and collections, named constants and types, booleans, enumerated types and character strings. However, it should be noted that these additional data structures of Spectur are based on those of Turing for which there exists a formal mathematical definition [Holt et al. 88].[2] This orientation toward programming language data structures is a key part of Spectur.

2.4 Turing and Spectur

It was decided that, to achieve our goals, this new specification language would be based on Turing, a programming language developed at the University of Toronto [Holt and Cordy 86]. This choice was particularly appropriate because Turing has a formal mathematical definition [Holt et al. 88],

[2] In particular, Turing/Spectur pointers are abstract data types and should not be confused with pointers of the C language.

is very expressive and readable, and is well understood by the authors. Turing can be described as a Pascal-like general purpose programming language, but with a cleaner syntax than Pascal and augmented by such features as modules, dynamic arrays and convenient string operations. It has been used with great success in a number of environments for several years.

The Turing Plus language is an extension to Turing. Turing Plus adds concurrency, exception handling and separate compilation as well as "dirty" and "dangerous" features such as type cheats and absolute memory addressing that facilitate system programming [Holt and Cordy 87]. While Turing Plus is a proper extension of Turing (*i.e.* any Turing program is also a Turing Plus program), Turing Plus has not been formally defined — its presumed knowledge of the underlying hardware would make a mathematical definition unwieldy.

Spectur was based on Turing proper, so as to take advantage of its formal definition. It is not yet clear how useful Spectur will be in specifying Turing Plus programs. Indeed, it is well known that concurrency alone presents a variety of problems for the specifier.

2.5 Spectur Specifications

It was decided that the appropriate level at which to attempt the formal specification of a program was the module level. The Turing module is a well-defined entity that consists of a set of data structures and a set of procedures, plus the user interface. A user can access the module only through this interface, usually by passing parameters to a procedure that has been "exported" by the module. Thus, a user can alter the data structures of the module only indirectly, by a procedure call.

The basic approach now seemed clear: the formal specification of a module should closely resemble the actual program module. It should have a similar syntax, and all externally-visible parts of the specification module — the module name, the import and export lists, and the names and formal parameters of all exported procedures — should be identical to those of the program module. The program module can then be specified by declaring abstracted data structures (chosen on the basis of descriptiveness rather than efficiency), and specifying the actions of the exported procedures via pre- and post-conditions on those data structures. This, indeed, was the approach chosen for Spectur.

3 An Example Specification: MiniTunis

To get a good idea of how Spectur might be useful, it was decided to attempt a non-trivial example specification. However, rather than inventing a complicated example, it was thought that it would be easier and just as interesting to specify an existing program (although this might seem a backward step in program development, creating a specification from an implementation is an example of reverse engineering [Chikofsky and Cross 90]). An operating system seemed a good choice, since these programs are invariably large and complicated. Indeed, the formal specification of operating systems is a popular research area, especially for systems in which reliability and security are critical.

3.1 Tunis and MiniTunis

The first major decision was the choice of the operating system to be specified. The first to be considered was Tunis (*T*oronto *Uni*versity *S*ystem) — an implementation of the UNIX[3] operating system written in Turing Plus [Holt 83]. The major design goal of the Tunis project had been to create a system that would attain the functionality of UNIX, but also be more reliable, portable, secure and semantically clean than most traditional C-language implementations.

However, because Tunis implements all of UNIX, it is quite large. It was decided instead to attempt a formal specification of MiniTunis, a much smaller UNIX-like system. MiniTunis, also written in Turing Plus, uses many of the design techniques employed in Tunis [Penny 87]. Although small (it has no directory hierarchy and implements only 13 system calls) and useful more as a teaching tool than as a practical operating system, MiniTunis is nonetheless a good model for larger systems, such as the full Tunis system. It was hoped that a successful specification of MiniTunis could be extended to Tunis.

[3] UNIX is a trademark of Bell Laboratories.

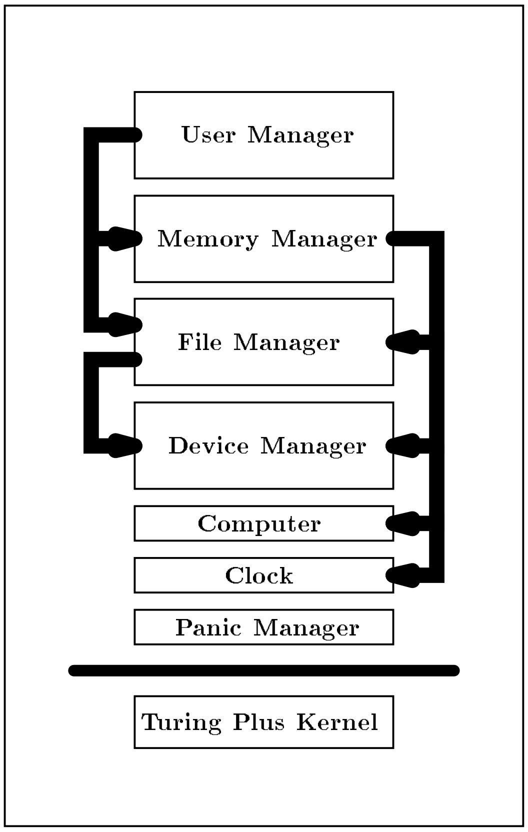

Figure 1: Module hierarchy of MiniTunis.

3.2 Choice of Module

The hierarchy of the major modules of MiniTunis is illustrated in Figure 1. Each box represents a major module, and each arrow represents an access by one module of another below it in the hierarchy (in addition, all modules access the Panic module, but these arrows have been omitted). The modules of MiniTunis are arranged in a semi-hierarchy — that is, any module may access any other module at a lower level. This approach is different from the traditional C implementation of UNIX, where there is no such rigid hierarchy of the modules.

The next question to be considered was how best to specify the actions of MiniTunis. The obvious starting point seemed to be at the system call level, since this was the top layer of the system. However, most of the system calls of the user manager were actually composed of calls to the file manager and to the memory manager. While it was felt that Spectur was capable of specifying the system calls, it was thought that incorporating the memory manager into the specification would involve a lot of complexity that would distract us from our central goal of learning about Spectur and its usefulness. For these reasons, it was decided that the file manager was a better choice for specification.[4]

3.3 "Sneak Paths"

It became obvious that since the file manager called on the device manager module, the device module would have to be implicitly incorporated into the specification of the file manager. This is one result of using a hierarchical module structure: a module has no knowledge of the modules above it, but (potentially) full knowledge of the ones below. Since the file manager makes frequent use of the system's devices, it seemed that the devices would have to be abstracted into the file manager with the tacit assumption that all device accesses must be done indirectly, via a call to the file manager. In a strict hierarchy, where a module may access only the module directly below it, this would be a safe assumption. But MiniTunis is a semi-hierarchy, and so it is possible that a module above the file manager could also access the device manager without going through the file manager. Indeed, the memory manager does just that — it performs I/O on one of the system's disks (the "swap" disk) directly through the device manager, rather than indirectly through the file manager. Thus, there is a problem: the specification of the file manager presumes that it has exclusive control over the system devices, but the memory manager bypasses the file manager to access the swap disk.

Unfortunately, there is not much that can be done about the existence of such "sneak paths" — with a semi-hierarchy they are unavoidable. It might be argued that a semi-hierarchical approach simply should not have been used. However, since the specification was done after the implementation, the design of MiniTunis could not be changed easily. Another possible solution would have been to specify the system as if it were a strict hierarchy — that

[4]Morgan and Sufrin have studied the similar problem of formally specifying the UNIX file system using a Z-like notation [Morgan and Sufrin 84].

is, assume that all accesses of modules below the file manager are done via calls to the file manager. This would have simplified the specification, but only at the cost of making it unlike the true system.

Fortunately, in the case of MiniTunis, the problem was easily solved with a weaker assumption: since the memory manager accesses only the swap disk directly, and since no user is supposed to access the swap disk through the file manager (except possibly for system debugging), the formal specification of the file manager simply will not guarantee the integrity of the file system if a write is done to the swap disk via the file manager. However, this assumption does make the specification less precise than it could be, for it does not explain what would happen should such a write to the swap disk occur. Unfortunately, a complete answer would involve unfolding all of the details of the memory manager, and it was decided that these details were not germane to the specification of the file manager.

3.4 The Form of the Specification

The next major decision concerned the form of the file manager specification. Since one of the major goals was to minimize the distance between program and specification, it was decided that the basic Turing programming paradigm of:

$$\textit{module} = \textit{data structures} + \textit{exported procedures}$$

should also be the underlying paradigm of the specification.

So the first challenge in specifying the MiniTunis file manager was to determine an appropriate abstraction of its major data structures. After much thought, four structures were identified: *file*, the files of the system (devices were considered to be special files); *directory*, the set of accessible file names (MiniTunis has a flat file system and thus has no directory hierarchy); *filesInUse*, a set containing references to all existing files (the directory plus the set of "zombie" files — files that have been unlinked but are still in use); and *openFileTable*, a table for referencing the open files. These data structures can be sketched pictorially (Figure 2) to capture our intuitive idea of the essential state of the system. Given this diagram, we can proceed with confidence to write a precise specification of these structures and the operations that modify them.

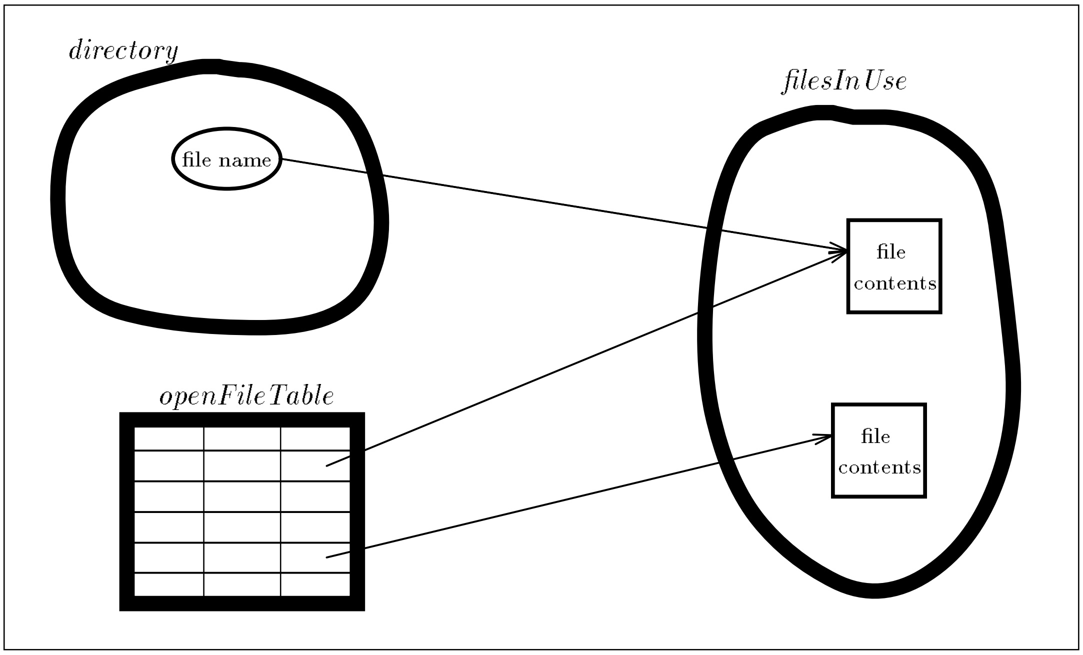

Figure 2: MiniTunis file manager data structures.

Figure 3 shows how a slightly simplified version of the data structures can be specified in terms of a collection (a set with allocable elements), two sets and a map. The figure also shows a (slightly simplified) specification of the **Seek** procedure using pre- and post-conditions. The other principal exported procedures — **Create**, **Open**, **Close**, **ReOpen**, **Unlink**, **Read**, and **Write** — can be specified in an analogous manner [Godfrey 88].

3.5 Concurrency

At first, it was decided to ignore the fact that MiniTunis is a concurrent program — the file manager was specified as if there were only one thread of execution at any time. The reason for this was to make the task of specification as straightforward as possible initially, as it is well known that the semantics of concurrent programs can be very complex. It was thought that it would be easiest to understand MiniTunis by first considering it to be a sequential program.

However, after an examination of the source code, it became apparent

```
module File

    export Create, Open, Unlink, ReOpen, Close, Seek, Read, Write

    var file : collection of
          record
             accessCount : nat              % "nat" is the natural number type
             contents    : map nat to Byte
             fileSize    : nat
             zombie      : boolean
          end record

    var directory       : set of FileName          := null

    var filesInUse      : set of pointer to file   := null

    var openFileTable : map FileNumber to
          record
             accessCount : nat
             filePtr     : nat
             f           : pointer to file
          end record
          .
          .             <declarations of other procedures>
          .
    % SEEK--Move the file pointer for the given openFileTable entry to the
    %       position indicated.

    procedure Seek ( fnum : FileNumber, position : nat )

    import % 'openFileTable' is the only module data structure referenced in
           % the Seek specification.  It is imported as 'var' because write
           % access is required.
           var openFileTable

    pre    % Check that the openFileTable entry is currently in use.
           openFileTable(fnum).accessCount > 0

    post   % Reset the filePtr field for this entry in the openFileTable.
           % The way the code is interpreted is:  all fields of the record
           % variable 'openFileTable(fnum)' have the same value as before,
           % except the field 'filePtr' which is set to the value 'position'.

           openFileTable(fnum) = [openFileTable(fnum)] ( filePtr : position )

           % No other entry in the openFileTable changes.  This code is
           % interpreted as:      unchanged(openFileTable(1))
           %                  and unchanged(openFileTable(2))
           %              ... and unchanged(openFileTable(FileTableSize))
           % omitting the index 'fnum' from the list.

           and for fn in FileNumber-{fnum} : and (unchanged(openFileTable(fn)))

    end Seek
          .
          .             <declarations of other procedures>
          .
end File
```

Figure 3: Outline of the file manager specification.

that the fact that the file manager was concurrent was not visible to the outside user. This is because the exported procedures of the file manager are *serializable* — that is, if several calls to the module are overlapping in time, the end result will be as if the calls had been arranged in some linear order and executed sequentially. The order in which these calls are apparently executed, however, is not predictable.

We gained considerable simplicity from the fact that the MiniTunis file system behaves in a serializable manner. We have avoided facing the problem of specifying non-serializable systems.

3.6 Level of Detail

As programs, operating systems traditionally have been very difficult to specify because of their complexity. Even MiniTunis, a very small and clean system, has a few implementation details that would cause an exact specification of its behaviour to be as complicated as the source code. Some of these details can be handled by making certain assumptions about its users. For example, as mentioned previously, it is assumed that no user will want to write directly to the swap disk; to describe all that would actually happen should a user write to the swap disk would necessitate the inclusion of all of the implementation details of the memory manager. The assumption is justified on the grounds that the actual system will fail to operate predictably if the user does arbitrary direct writes.

Similarly, sometimes a procedure call such as **Create** or **Open** is unable to execute normally because there is insufficient memory left in the system. When this happens, the procedure simply sets a flag and returns, cleaning up any mess it may have created in the attempted execution. The file manager formal specification explicitly allows for the possibility of resource exhaustion, but is deliberately vague about when it may occur — in effect, it says "Either the call executes normally, or nothing happens", and is not explicit about which will be the case. To be able to predict exactly when a procedure call will run out of space would involve keeping track of the allocation of memory space by the particular machine. It seems that this level of detail is not appropriate for a specification, since another implementation of MiniTunis would likely run out of resources in a slightly different way. Instead, the specification should allow for the possibility of non-deterministic resource exhaustion without overly constraining the way in which an implementation uses machine resources.

In general, the specifier must decide what level of detail is appropriate for the specification. A byproduct of the inherent semantic complexity of operating systems is that they often, either directly or indirectly, exhibit traits that are due either to the particular implementation of the code, or to the hardware and/or software on which it runs. A very detailed specification that includes these traits may, as a result, be excessively constraining and difficult to read.

3.7 Observations

The original goal of this experiment was to formally specify a UNIX-like operating system. What has been accomplished, with the use of the Spectur language, is the specification of a large portion of a simple system — the file manager of MiniTunis. Certain assumptions were made, but these simplifications have led to a fairly elegant specification.

In specifying MiniTunis, it was found that the Spectur language is easy to use, especially in conjunction with the Turing (Plus) programming language. Spectur is also easy to learn, since the potential specifier need be familiar only with Turing data structures and pre- and post-conditions. The gap between program and specification is minimized, since there is a great overlap between Turing and Spectur data structures and because the overall syntax is similar. Spectur is also mathematically precise, which makes it attractive for use with highly reliable or secure operating systems. It is felt that, based on the success in specifying the MiniTunis file manager, the specification of a larger, more practical operating system such as Tunis is possible using the Spectur approach.

4 Conclusions

4.1 How Spectur is Useful

There are a few restrictions about using Spectur to specify other programs. First, the program must be a Turing (or Turing Plus) module — no attempt has yet been made to make Spectur compatible with other implementation languages such as C or Ada. Second, if the program allows concurrency, then the actions of the module interface must be serializable. Finally, it should be mentioned that as yet no software exists to support Spectur specifications. However, our experience in writing Turing software suggests that, at the very least, the creation of a type checker would be straightforward.

4.2 Spectur and Other Specification Languages

Other specification languages have extensive software tools. Some, such as Z [Spivey 88] and VDM [Jones 90], use a highly mathematical notation, which makes their specifications appear more like logic theorems than programs; this makes such specifications difficult to create and manipulate using a standard keyboard. Also, Z's use of exclusively mathematical data structures makes it less practical for use in the later stages of program development, where the specifications become more concrete (and less mathematical) in nature. Spectur has both mathematical and programming language-style data structures, which makes it convenient for use at all stages of program development.

Larch [Guttag et al. 85] uses a two-tiered approach to specification. The bulk of the work is done using the Larch Shared Language, which uses an algebraic approach to define "traits" that model the function of the program. The program interface specification is then done using the appropriate Larch Interface Language (such as Larch/Pascal). The interface specification describes the behaviour of the program in terms of the traits defined in the Shared Language specification, but using a state-based approach. The interface specification is usually very short; it provides an important link between the program and the Shared Language specification, while at the same time isolating the implementation language dependencies from the latter. Spectur differs from Larch in using only the state-based approach, which is felt to be easier to understand than the algebraic approach.

4.3 Spectur is Easy to Use

We will conclude this paper with an analysis of how well Spectur satisfies its primary design goal: ease of use. In particular, we will examine from four different perspectives: that of the specifier, the implementor, the user and the verifier.

A specifier/software designer who is attempting to create a specification of a proposed program from an informal description will desire a specification language that is expressive, powerful, and able to model informal descriptions conveniently. Spectur has the mathematical data types that a specifier would find most natural in representing an abstract system. Spectur's state-based approach also conforms closely to the commonly held mental model of what a program is: a black box whose behaviour can be described in terms of inputs and outputs. Currently, Spectur lacks the software tools that a software designer would desire, but we feel that these will not be difficult to construct.

An implementor who is designing a program (or a more concrete specification) from a formal specification will want the specification to resemble the desired program without overconstraining it. The overall structure of the specification should be similar to that of the desired program, but leave the "details" of the implementation to the discretion of the programmer. Spectur satisfies this desire by allowing the specification to be similar to the eventual program: both the specification and the program must be modules, have the same name and have identical entry points. Spectur has programming language-style data structures that suggest (but do not insist) how the program's data structures might be chosen. The use of pre- and post-conditions to specify procedures neatly conveys their effects while allowing the implementor the freedom to decide just how to satisfy the given constraints.

A user reading a specification as documentation will desire that the resemblance between it and the actual program be strong. Since a user will probably form his or her mental model of a program based on its observable effects, the state-based approach seems the most intuitive. Also, the great syntactical similarity between the specification and the program, as well as Spectur's use of programming language data structures, will help to convince the user that the specification really does model the implementation.

A mathematician attempting to prove program correctness from a formal specification will desire that the specification language have a mathematical definition, plus a mechanism for proving correctness. Spectur currently lacks a mechanism for proving correctness — this is a future research area. Because Spectur uses the state-based approach, it is not as convenient for proofs as specification languages that use an algebraic method.

The fundamental strength of the Spectur approach to specification resides in providing a notation that is immediately accessible to programmers. This has many advantages in the production of actual programs. It allows a person who is a competent programmer to produce a formal specification without a good deal of training. It gives the implementor immediate and comfortable access to the formal specification. Finally, it gives the programming-competent user an immediately accessible definition of the system. For these reasons, it is felt that the Spectur language is a useful addition to the study and practice of formal specification.

References

[Chikofsky and Cross 90] E. J. Chikofsky and J. H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, vol. 7, no. 1, pp 13-17, January 1990.

[Godfrey 88] M. W. Godfrey, *Toward Formal Specification of Operating System Modules*, M.Sc. thesis, University of Toronto, June 1988.

[Guttag et al. 85] J. V. Guttag, J. J. Horning, and J. M. Wing, "Larch in Five Easy Pieces", Technical Report 5, Digital Equipment Corporation Systems Research Center, July 1985.

[Hoare and Wirth 73] C. A. R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language Pascal", *Acta Informatica*, vol. 2, no. 4, 1973, pp 335-355.

[Holt 83] R. C. Holt, *Concurrent Euclid, The UNIX System and Tunis*, Addison Wesley, 1983.

[Holt and Cordy 86] R. C. Holt and J. R. Cordy, "The Turing Language Report", Technical Report, Computer Systems Research Institute, University of Toronto, August 1986.

[Holt and Cordy 87] R. C. Holt and J. R. Cordy, "The Turing Plus Report", Technical Report, Computer Systems Research Institute, University of Toronto, September 1987.

[Holt and Cordy 88] R. C. Holt and J. R. Cordy, "The Turing Programming Language", *Communications of the ACM*, vol. 31, no. 12, December 1988.

[Holt et al. 88] R. C. Holt, P. A. Matthews, J. A. Rosselet, and J. R. Cordy, *The Turing Programming Language: Design and Definition*, Prentice-Hall, 1988.

[Jones 90] C. B. Jones, *Systematic Software Devleopment Using VDM*, Second Edition, Prentice-Hall, 1990.

[Meyer 85] B. Meyer, "On Formalism in Specification", *IEEE Software*, vol. 2, no. 1, January 1985, pp 6-26.

[Morgan and Sufrin 84] C. Morgan and B. Sufrin, "Specification of the UNIX Filing System", *IEEE Transactions on Software Engineering*, vol. 10, no. 2, March 1984, pp 128-142.

[Neumann 80] P. G. Neumann, "Experiences With A Formal Methodology for Software Development", Technical Report CSL-120, SRI, October 1980.

[Penny 87] D. A. Penny, "MiniTunis Code", private circulation, 1987.

[Spivey 88] J. M. Spivey, "An Introduction to Z and Formal Specifications", in Michael Spivey *The Z Notation: A Reference Manual*, Prentice-Hall International, November 1988.