Science of Computer Programming 2 (1982) 53–68
North-Holland Publishing Company

SOME NOTES ON PUTTING FORMAL SPECIFICATIONS TO PRODUCTIVE USE

John GUTTAG
MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, U.S.A.

Jim HORNING
Xerox Palo Alto Research Centers, 3333 Coyote Hill Road, Palo Alto, CA 94304, U.S.A.

Jeannette WING
MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, U.S.A.

Communicated by M. Sintzoff
Received February 1982
Revised June 1982

Abstract. These notes are personal reflections, stemming from attempts to understand the sources of problems and successes in the application of work on formal specifications. Our intent is to provoke thought about the nature and value of work in the area; not to provide a set of well-tested results. Rather than focusing on yet another specification language, we have tried to take a broad view of the role of formal specifications in the program development process.

1. Introduction

To treat programming scientifically, it must be possible to specify the required properties of programs precisely. Over the last few years much has been published about specifications in general, and formal specifications in particular. Most of these papers have presented or evaluated particular specification languages or classes of specification languages. Here we try to take a somewhat broader view of the role of formal specifications in the program development process.

These notes are personal reflections, stemming from attempts to understand the sources of problems we encountered in the application of our earlier work on formal specifications. Our intent is to provoke thought about the nature and value of work in the area, perhaps to open a dialog, but not to provide a set of well-tested results.

Formality is certainly not an end in itself. The importance of formal specifications must ultimately rest in their utility – in whether or not they are used to improve the quality of software or the cost of producing and maintaining software. To date, informal or semi-formal techniques have been far more widely used than have formal ones. Undoubtedly, they have been of significant help.

As our understanding of the theoretical and linguistic aspects of formal specifications improved, we began to try to use them in developing interesting software. We ran into problems doing this, and were eventually forced to conclude that for the most part our use of formal specifications had not helped us either to produce better software or to produce software more efficiently. The difficulty was not that we encountered things that were intrinsically difficult to specify, but that

(a) We found it difficult to establish a useful correspondence between our elegant formal systems and the untidy realities of writing and maintaining programs.

(b) We were unable to maintain a consistent attitude about what constituted a 'good' specification. This confusion about how to evaluate both what we were doing and the results of what we were doing often kept us working in circles.

(c) We found ourselves getting bogged down by the clerical details involved in managing, reasoning about, and maintaining the consistency of large specifications.

Despite all of this, we remain optimistic about the future utility of formal specifications of programs. There are many problems, but none of them seems intractable. We believe that eventually formal specifications will supplant informal ones as the tool of choice in many areas.

2. Some vocabulary

One symptom of our problems was our use of the same words to mean a number of different things. So we begin by introducing some terminology.

2.1. Definitions

A (formal) *specification language* consists of two sets, *Syn* and *Sem*, and a relation, *Sat*, between them. The first set is called its *syntactic domain*; the second, its *semantic domain*; the relation is called *satisfies*.

For a specification language, if *Sat*(*syn*, *sem*), then *syn* is a *specification* of *sem*, and *sem* is a *specificand* of *syn*.

The *specificand set* of a specification is the set of all its specificands.

A specification is *satisfiable* (or *consistent*) if its specificand set is non-empty.

Somewhat less formally:

A *specification language* provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects *satisfy* each specification.

A *specification* is a sentence of a specification language. It denotes a subset of the semantic domain, called its *specificand set.*

A *specificand* is an object *satisfying* a specification.

The *satisfies* relation provides the semantics of the specification language. Note that different specification languages may have identical syntactic and/or semantic domains, but different satisfies relations.

BNF is an example of a simple specification language, with a set of grammars as its syntactic domain and a set of strings as its semantic domain. Every string is a specificand of each grammar that generates it. Every specificand set is a formal language.

2.2. Semantic domains for program specification

The choice of a specification language's semantic domain is a matter of some interest. 'Abstract data type specification languages', for example, have been used to specify algebras [5, 26], theories [3], and programs [7]; programs have been used to specify functions from input to output [19], computations [22], predicate transformers [4], and even machine language instructions. In this paper, we confine our attention to specification languages whose specificands are programs or systems of programs. I.e., each specificand is software that *implements* its specification.

Quite a bit can be said about the power and limitations of a specification language without any consideration of its syntactic domain, simply by considering possible functional decompositions of its satisfies relation. Henceforth, we will associate with each specification language a homomorphism, A, which we call a *semantic abstraction function*, that maps elements of the semantic domain into equivalence classes. The semantic abstraction function will be chosen so that there exists an induced relation, ASat, which we call an *abstract satisfies relation*, such that,

$$\forall spec \in Syn,\ prog \in Sem\ \ [Sat(spec, prog) = ASat(spec, A(prog))].$$

We will call *range*(A) the *intermediate domain* of the specification language. Programs that A maps to the same equivalence class in the intermediate domain cannot be distinguished by specifications written in its associated language.

Different semantic abstraction functions make it possible to impose different kinds of constraints on programs. It can be useful to have several specification languages with different semantic abstraction functions for a single semantic domain. This encourages complementary specifications of different aspects of a program.

2.3. Some distinctions

We find it helpful to classify specifications – and the languages in which they are written – in a variety of ways. The point of this section is not that these distinctions introduce sharp dichotomies, nor that they provide a complete taxonomy. Rather, it is that each of them has important consequences on the development, use, and evaluation of specifications.

Class of constraint. A specification language's semantic abstraction function limits the class of constraints that it can impose on programs. Suitably chosen, it can simplify both the language and specifications written in the language. We distinguish two broad classes of semantic abstraction functions (hence, of program specification languages and of specifications): those that abstract preserving each program's *behavior*, and those that abstract preserving each program's *structure*.

Specifications written in *behavioral* specification languages indicate constraints only on the observable behavior of their specificands. The required functionality (mapping from inputs to outputs) of any specificand is an important behavioral constraint. However, behavioral specifications may also constrain other aspects of behavior, e.g., response time, space efficiency, or reliability.

Specifications written in *structural* specification languages indicate constraints on the internal composition of specificands. Divide-and-conquer is a fundamental method of programming, and the specification of decomposition is sufficiently well understood that several programming languages incorporate structural specifications [2, 13]. The combination of a rigorous structural specification of a system with informal behavioral specifications of its components is quite common [24, 10].

It is easy to confuse a specification of structure (of specificands) with the structure of a specification. Any large specification will need some structure of its own, as an aid to dealing with the specification itself. This does not necessarily correspond to any structure in the specificands. The overriding goal in decomposing a specification is clarity, whereas there are many other important constraints on the decomposition of a program.

Viewpoint. Useful programs are not closed systems. An unobservable program is of no interest. Behavioral specifications rely on the notion of *observability*. What is considered observable forms the *boundary* (or interface) of the specificand.

A programming language is a convenient semantic domain partly because it provides a standard definition of what may be observed (e.g., values of non-local variables, input/output, exceptional conditions). Constraints on programs can be stated in programming language terms. We call such specifications *language-oriented.*

By contrast, clear expression of some important kinds of constraints on a system may require a viewpoint outside the programming language. For example, the significant behavior may involve external devices whose behavior is controlled or interpreted by humans, and may best be described in terms of abstractions derived from the application domain. Finding good abstractions for these descriptions is an important research area [21, 7]. We call specifications based on such abstractions *application-oriented.*

Environment. For any boundary, there is a duality between a system and the 'environment' that provides its inputs and (presumably) consumes its outputs [9]. The role of system and environment can be reversed by exchanging the roles of inputs and outputs. When the environment is itself a system amenable to formal specification, it is possible to exploit this duality. *Assumptions* about the environment may simplify the statement of the *constraints* on the system; e.g., it is frequently simpler to specify a pre-condition on the execution of a procedure than to specify its behavior for arbitrary inputs. To ensure validity, such assumptions must become part of the specifications of that environment.

Specification size. Specifications, like programs, come in a great range of sizes. The processes of writing, reading, and checking large specifications differ in important ways from those for small ones. Languages and methods that were adequate for our small specifications failed to scale up well when we attempted larger specifications. It is not fruitful to try to define a precise dividing line between large and small, but when the text of a specification exceeds a few pages, problems of scale begin to dominate.

There is no necessary connection between the size of a specification and the size of its specificands. A specification can have specificands of very different sizes, and a single program can satisfy specifications of very different sizes. In our experience, however, there has been a definite correlation between the size of a specification and the specificand we have implemented.

System, local, and organizational specifications. Three combinations of attributes are so common that we have found it convenient to name them. We call application-oriented behavioral specifications of (typically large) collections of programs *system specifications.* They express constraints on a system in terms of what can be observed by its users.

We call language-oriented behavioral specifications of single program units *local specifications.* They are typically small, and are the kind of formal specification that appears most often in the literature, e.g., specifications of stacks, sets, and factorial functions. They express constraints on a program in programming language terms. For example, if the programming language allows for exceptional termination of a procedure, procedure specifications should include conditions for both normal and exceptional termination.

System and local specifications are both behavioral, and it is tempting not to distinguish between them. However, their differences are probably more important than their similarities.

One key difference is the process of formalizing observable behavior. What is required by users of a system is not necessarily more complex than what is required by callers of a procedure. But the burden of formalizing the observable state of a programming language falls on the specification language designer, while the burden of formalizing the observable state of a system generally falls on its specifier.

Formalizing the aspects of system behavior that are to be constrained is often hard. Much of the complexity of Sufrin's text editor specification occurs here [20]. Sometimes, however, it is not so difficult: Some successful design methodologies are based on the observation that a few application-oriented abstractions suffice for the description of large classes of useful systems with very stylized observable behavior, e.g., doing all I/O to highly-structured files [10]. Finding and formalizing these abstractions is one of the best ways of converting 'complex' specification problems into 'simple' ones.

A second important difference is size. Useful systems often have more different (and more complex) kinds of behavior to be specified than individual program units

do. Hence, system specifications tend to be much larger than local specifications. The obvious analogy is valid: The process of constructing system specifications is related to the process of constructing local specifications, as the process of constructing systems is related to the process of constructing program units.

Finally, we call the combination of a structural specification with a behavioral specification of each of the system's components an *organizational specification*. It should be possible to demonstrate that an organizational specification implements a system specification – even before completing the design of the components – by showing that the system will satisfy its specification if each component satisfies its specification.

3. Some questions

Several questions have helped us to focus on key issues connected with different uses of specifications: What is accomplished by constructing them? What benefits result from their existence? When should they be written? By whom should they be written? By whom should they be read? By which properties should they be evaluated?

In trying to answer these questions it became apparent to us that there is not a single set of 'right' answers. In this section, we will discuss a range of answers relevant to the various kinds of specifications discussed above.

3.1. What is accomplished by constructing them?

We have often profited more from the process of constructing a specification than from the use of the completed result. The process of formal specification encourages prompt attention to inconsistencies and incompleteness and ambiguities in understanding. Each of our efforts in program specification has clarified our understanding of the specificand set – whether the attempt came before or after the construction of the program. In many cases, such improved understanding has been the major result of the specification effort.

The construction of system specifications focuses attention on what is required of the system's behavior. It serves as a mechanism for generating questions that should be answered by the client, rather than the implementers [7]. If questions are posed early in the system life cycle, they are more likely to be answered by the client or the system designer, rather than by lower-level programmers. The construction of local specifications systematically focuses attention on implementation decisions that remain to be made. The construction of structural specifications focuses attention on the decomposition of a system into subsystems. The construction of organizational specifications focuses attention on the suitability of such decompositions.

3.2. What benefits result from their existence?

Once written, specifications can serve many different purposes during the software life cycle. They provide a tangible record of the understandings that were acquired during their construction – a record that can be helpful to clients, designers, implementers, maintainers, and users.

System and local specifications state an agreement between providers and users of a service. A good system specification can provide guidance for system implementers, reduce disputes between vendors and clients, and perhaps serve as the basis for a legal contract. Local specifications make it possible to separate consideration of the implementation and the use of a program unit. They provide the 'logical firewalls' that permit divide-and-conquer to succeed.

A structural specification provides a decomposition of a system into units that can be specified, designed, implemented, tested, and evaluated separately. An organizational specification can play the part played by a model in other fields of engineering design. It can both guide and document the design process. The precision with which the components are specified will affect the success of attempts to deal with them independently, and the likelihood of their combination meeting the requirements of the system as a whole. A good organizational specification can enable meaningful pre-implementation evaluation of the design, and provide unambiguous guidance to implementers.

3.3. When should they be written?

It is generally useful to start writing specifications as soon as the decisions they record have been made. Intentionally delayed decisions can be recorded as parameters in the specification; they should not be confused with accidental incompleteness or ambiguity. Since specifications only become irrelevant when the program has ceased to be useful, it is a mistake a treat the process of writing specifications as separate 'phase' of a software project that terminates before the project itself. Specifications should evolve as the program itself evolves.

The first version of the system specification should generally precede the design of the system itself. Structural specifications should be developed as the designers partition the system into subsystems; later they can be converted to organizational specifications, by writing local specifications for their components. Local specifications for general-purpose components should be written before they are placed in a library.

We have had some experience with the *ex post facto* construction of specifications of existing programs. In addition to providing a document that can be useful to maintainers of the software, such an exercise can lead to useful generalizations. Concepts developed for a particular system (and lessons learned through its construction and use) are often more general than the system itself. To communicate and reuse such concepts, it helps to abstract and specify their essential characteristics.

3.4. By whom should they be written?

We do not expect good specifications to be written by unskilled persons. The real question is: What skills are needed to write various kinds of specifications?

System specifications could be written by specialists who communicate with implementers on one side and clients on the other. Alternatively, they could be written collaboratively: "It is unrealistic to expect the software developers to have enough expertise to check the specification against the intended application domain, let alone carry out the abstraction process. On the other hand, it is plain wishful thinking to assume that experts in an application domain will provide us with a theory that would have all the properties necessary.... The most abstract specification must, therefore, be constructed in co-operation between a programmer and application expert (client)." [21]

It seems clear that structural and organizational specifications should be written by system designers. Local specifications may be written by system designers (for systems designed top-down) or by implementers (for bottom-up programs, such as library routines). Both groups may benefit from specialist assistance, but they shouldn't try to delegate the responsibility for this task.

3.5. By whom should they be read?

Since more people will read them than write them, we expect that the readers of (at least some kinds of) specifications will be less well trained than their authors. Nevertheless, some skill is required for any formalism to be intelligible.

It is crucial that a system specification capture the client's requirements. Unfortunately, those who best understand the problem may be ill-equipped to read and understand such specifications. A specification specialist may have to serve as guide and interpreter. It is equally important that the system's designers, implementers, and maintainers understand this specification, but we believe that they are more likely to have the training required to do so.

Local, structural, and organizational specifications are read by designers, implementers, and maintainers, but are not intended for clients and users.

3.6. By which properties should they be evaluated?

It is important to keep in mind the difference between evaluating a specification and evaluating some or all of its specificands. We ourselves have not always been as careful about this as we should have been. When we were evaluating small specifications it didn't much matter. The most interesting property of a small specification is often its specificand set. For larger specifications, however, there are other properties that are too important to ignore. The distinction between specification and specificand is especially important when discussing properties that make sense in both domains. A simple example is the difference between an ambiguous grammar and an ambiguous sentence.

Specification properties. It is desirable for any specification to be syntactically well-formed, and consistent. Specifications need not be 'complete' in the logical sense, but they should not conceal oversights. More amorphous, but equally important, problem-independent qualities such as lucidity, modularity, precision, and concision, are much harder to deal with; as is the problem-specific quality of "saying what the client requires".

Properties of specificand sets. Much of the evaluation of particular specificands takes place after they are implemented. However, given a specification, it may be useful to evaluate the properties of its specificand set, or of some or all of its members, based on the information in the specification itself. Are there any small programs in the set? What bounds can be placed on the efficiency of specificands? For a given class of 'small' changes in the specification, are there specificands that can satisfy the changes with correspondingly 'small' changes?

In [7] we suggested a way of testing the correspondence between system specifications and the client's intentions. It is based on stating interesting problem-specific conjectures, and attempting to prove them as theorems that follow from the specification. We do not have nearly enough experience with this method to draw any firm conclusion (except that it is naive to rely on our own accuracy as theorem-provers), but we are guardedly optimistic.

Evaluation. In evaluating specifications, specifiers are limited by the lack of precise definitions for most of the interesting problem-independent properties [23]. They fall back on intuition, programming experience, and informal arguments. We are currently trying to develop formal definitions for a variety of interesting problem-independent properties of behavioral specifications. We hope to clarify properties of useful specifications and provide some helpful rules of thumb for specifiers. We are looking at both those evaluation criteria that appear in the literature on formal specifications, e.g., implementation bias [11] and well-definedness [12], and also at more informal guidelines for program design [17, 15].

4. Tools

It is a poor workman who blames his tools, but it is a foolish workman who does not appreciate good tools. The task of specification will remain a significant intellectual challenge, even with the best tools we can foresee. However, our current primitive tools are a significant bottleneck in the application of formal specifications to program development. We place tools in two general categories: mental tools and 'metal' tools. Over the last few years we have been concerned primarily with the development of our mental tools.

Mental tools include specification languages, methods (global strategies), techniques (local tactics and tricks), and, most importantly, experience. The value of experience is brought home strongly whenever we teach someone to write formal specifications. The specification language we use can be taught in a day, but we

find that it takes weeks for even a good student to become competent, and months to attain proficiency. The difficulty of learning to write good specifications is comparable to the difficulty of learning to write elegant programs. It has been a serious limitation in spreading the usage of formal specifications.

Recently we have turned our attention to the development of software (a relatively new kind of metal) tools that we hope will help with this problem. We expect this work to augment our mental tools in several ways. Firstly, developing tools to help with a task is complementary to studying the mental processes used in carrying it out. The design of various programming languages, for example, has both influenced, and been influenced by, the study of programming methodology.

Secondly, if we are successful in constructing software tools that encourage people to approach the tasks of writing and using formal specifications in the ways we find to be most effective, they will help to teach them our mental tools. We hope not only to impose some structure on the way they approach these tasks, but also to provide a mechanism to pass on much of the knowledge we have gained through experience.

Thirdly, a good set of software tools will allow specifiers to concentrate their energy and ingenuity where it is most necessary. Both expert and novice users can be spared the trouble of performing the substantial amount of routine work involved in writing, checking, and reading specifications.

There are dangers inherent in building and using such tools. A technique may achieve unwarranted permanence by being enshrined in a body of code; bad tools can lock us into unfortunate ways of doing things. Considering the limited body of experience with formal specifications, it is inevitable that early tools will incorporate some bad ideas. However, we need to accumulate experience using various tools to enable us to distinguish the truly helpful from the merely plausible.

4.1. Specification languages as mental tools

A reasonable specification may constrain many different aspects of its specificands. Even a 'simple' sort procedure might be required to satisfy a large number of rather different constraints: termination in $O(N \log N)$ time, bound of $O(N)$ working storage, establishment of a permutation of its parameter in ascending order by key, idempotence, stability, maintenance of the permutation invariant outside atomic swap operations, no more than $O(N)$ page faults with fixed working set, etc.

A single formal specification (written in a 'universal' specification language) incorporating precise versions of so many kinds of constraints is unlikely to be either brief or easy to understand. It seems more reasonable to use several languages with a common semantic domain (the programming language), and intermediate domains appropriate to the different kinds of specification. Then a program or system *satisfies a collection* of specifications in these languages if it satisfies each of them. Equivalently, the *specificand set of the collection* is the intersection of their specificand sets.

A full-fledged family of specification languages would contain languages for structural specification, system specification, functional specification, performance specification, reliability specification, etc. Several of these might include variants for different programming languages – although we expect a large degree of commonality among such variants.

Both the desire to consider various types of constraints separately, and the need to find some way to structure the presentation of large specifications suggest that 'putting specifications together' is a key issue for specification languages [3, 16]. It is important that the meaning of combinations be well-defined. It is also essential that the consequences of combinations be understandable to those who work with them.

We are presently engaged in the design and testing of a family of behavioral specification languages, which will be described in a future paper. Each member of our family has a component derived from a programming language, and another component common to them all. We call the former the *interface languages*, and the latter the *shared language*.

Each of our interface languages deals with what can be observed about the behavior of programs written in its programming language. Its simplicity or complexity is a direct consequence of the simplicity or complexity of the observable state and state transformations of the programming language. The shared language is more closely akin to traditional mathematics, and allows us to specify abstractions useful for stating constraints in a language-independent way. Its role is similar to that of *abstract models* in some other styles of specification.

It is our hope that most of the effort involved in writing a specification can be invested in the shared language component. The part written in an interface language should deal only with programming language dependent issues, e.g., calling sequences and exceptions. The invention and description of key abstractions should be done in the shared language. One reason for separating the two language components is that we expect common abstractions to be useful in specifying interfaces in many different languages. Some of them will be developed for particular applications; a few central ones will be useful in many applications. We plan to accumulate a library of reusable shared-language specification components and to include mechanisms in our shared language for customizing them.

4.2. Some experience with two existing software tools

We have experimented with the use of two very different tools, PIE [6] and Affirm [14], in constructing modest sized algebraic specifications.

PIE was designed as a Personal Information Environment. It is a very general database management system that includes facilities for creating, updating, querying, and browsing very flexible databases. It provides convenient facilities for multi-person sharing, multiple versions, and multiple views of the data. It can also be used to create a *meta* database of information about another database, allowing its use to be tailored to particular applications.

We used PIE to create a meta database of information about the structure and presentation of algebraic specifications and to create a database containing the specifications of some 30 abstract data types relevant to a system we were trying to specify. It was convenient to 'navigate' in the specification space, moving from the occurrence of a function or type in a specification to its own specification. It was also easy to display various 'projections' of the specification. The system automatically ensured various forms of syntactic consistency. We used it enough to become convinced of its potential as a tool for organizing specifications, developing them in the face of changing requirements or understandings, and for reading specifications.

Affirm was built as a program verification system. It provided only the most primitive facilities for structuring and perusing large amounts of information. It did, however, allow us to deal with the semantics of the specifications we wrote. Unlike many other verification systems, it emphasized the interaction between specification and verification. Using Affirm we wrote many algebraic specifications, checked them for logical consistency, checked them for various properties related to completeness, and proved theorems that followed from them.

Neither tool was completely satisfactory. Specifications in Affirm became masses of unstructured sets of equations. These were very hard to read and maintain, particularly when they had been constructed by someone else. Using PIE, on the other hand, we were able to put a great deal of structure into our specifications. Unfortunately, our PIE database treated the nodes of this structure as little more than strings of characters, and we greatly missed the semantic checking and theorem proving facilities that Affirm provided. We were not able to perform these tasks accurately without machine aid. We concluded that, at least for modest sized specifications, it is probably easier to maintain a manual substitute for PIE's structural information than it is to perform the equivalent of Affirm's checking manually.

4.3. Some software tools that we want

Just as the use of computers has radically revised conceptions of the amount of computation it is feasible to do, it is enlarging our perception of the amount of complexity it is feasible for a person to bring under intellectual control. A primary benefit of using a specification language that is formal is the ability to reason mechanically both about the well-formedness of specifications and about properties of specificands. We believe that the utility of any good specification language would be greatly enhanced by each of the following tools:

Syntax and 'type' checker. Experience indicates that most specifications (like most programs) contain 'errors' when they are written. Often these are typos or other careless mistakes, but sometimes they reflect serious conceptual problems. Fortunately, a surprisingly large number of the deeper problems have superficial symptoms, e.g., type errors. A tool that detected these symptoms early in the

specification process would be very helpful, in much the same way as a compiler's syntax and type checkers.

Semantic checkers. Mechanical theorem proving has progressed greatly in the last few years. If a specification language is designed with theorem proving in mind, a surprising amount of useful semantic checking can be done. Specification languages should be designed to encourage specifiers to inject a considerable amount of checkable semantic redundancy into their specifications. It is true that many interesting properties, e.g., consistency, are undecidable. Nevertheless, there are often useful and decidable sufficient conditions that guarantee the presence of a property, or necessary conditions that can warn of its absence. In addition to routinely performing checks relevant to the specification language, the theorem prover should be conveniently accessible to those who wish to prove or disprove conjectures about the specificands denoted by a specification.

Library. When faced with a new problem to specify, veteran specifiers rely on their past experience, borrowing ideas and components from previously written specifications. One of the great problems of neophyte specifiers is that they don't have such a repertoire to build on. A library of specifications that can be tailored to particular users and applications should prove helpful, *if* it can be organized so that relevant entries can be identified and retrieved with less effort than would be required to recreate them. Database management and information retrieval techniques may be relevant.

Editor. An editor that is tailored for specifications can help to write specifications [25]. It can supply templates for specifications, automatically generate redundant information about a specification or set of specifications, keep track of missing information (e.g., the type signature of a function name) and inconsistencies that must be fixed to make the specification well-formed. Careful design will be required to make these interactions efficient and pleasant.

Viewer. To help read specifications, a viewer, in conjunction with the library and retrieval tools, should let the reader see selected information about a particular specification or set of specifications. Results of various projections, combinations, expansions, and other transformations (both syntactic and semantic) on specifications should be displayable. The theorem prover should be available to test hypotheses.

5. Prospects

It is currently impractical to write complete formal specifications of all the important aspects of most useful software systems, just as it is generally impractical to construct complete formal proofs of system 'correctness'. Some have taken this as an indication of the impracticality of formal approaches to programming. We disagree. Many of the problems we have discussed plague attempts to create precise specifications in informal languages: often they are conjoined with greater verbosity

and less mechanically checkable redundancy. There are some parts of the program development process where various formal techniques are likely to be useful in the near future, and others where they are not.

Basic research is not the bottleneck in the area of local specification. With a modest investment in the development of some software support tools to help in the construction and analysis of local specifications and a somewhat larger investment in the education of system designers and implementers, it should be practical for professional programmers to write and use local specifications in the near future.

The practical application of formal system specification languages seems farther in the future. Experience with semi-formal approaches, as in the A-7 project [8], has illustrated both some of the potentials and some of the problems in this area. One problem is size. We need to get better at putting specifications together, at deriving the consequences of such combinations, at analyzing partially complete specifications, and at managing large amounts of formal information. A second problem is that, to be most useful, a system specification must be understood both by software designers and by clients who understand the intended application. It is likely that the early successes will be in areas where formalizing a few application-oriented concepts will suffice to describe many systems, or where the costs of imprecise specifications are considered unacceptable.

Structural specifications have been in productive use for some time. Various programming languages incorporate formal structural specification sublanguages. The construction of tools to take advantage of such specifications is an active area of research [18].

An organizational specification combines structural and local specifications. Our success in constructing them has been largely a function of our ability to design our software as a composition of logically self-contained modules with simple specifications. These specifications can be used to guide system construction. We would like to use them to evaluate the design of a system prior to starting an implementation, but we must still find practical ways to do this. However, their greatest importance may lie in precisely recording and communicating key design decisions.

In conclusion: Never let a problem with formalizing *all* relevant constraints on a program discourage you from stating as many as you can formally. The fewer that are left informal, the fewer the opportunities for misunderstanding or imprecision. Once a 'difficult' constraint has been separated from the others, it may be easier to see how it can be formalized in isolation.

Acknowledgment

Our views on the utility of formal specifications have been shaped over a long period by discussions with colleagues too numerous to list. IFIP Working Group 2.3 (Programming Methodology) and the informal DARPA working group on

Quality Software for Complex Tasks have both been very influential. Many of these ideas were crystallized by a workshop in Aarhus, Denmark [1]. Rod Burstall, Edsger Dijkstra, Bill McKeeman, Susan Owicki, David Parnas, Bob Ritchie, Mary Shaw, Wlad Turski, and two anonymous referees all contributed specific suggestions concerning various drafts of these notes.

This work was supported in part by the Office of Naval Research contract with DARPA funding #N00014-75-C-0661.

References

[1] *Proc. International Workshop on Program Specification*, Aarhus (Springer, Berlin, 1981).
[2] *The Programming Language Reference Manual*, Lecture Notes in Computer Science **106** (Springer, Berlin, 1981).
[3] R.M. Burstall and J.A. Goguen, Putting theories together to make specifications, *Proc. 5th International Joint Conference on Artificial Intelligence*, Cambridge, MA (1977) 1045–1058.
[4] E.W. Dijkstra, Notes on structured programming, in: O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare, *Structured Programming* (Academic Press, London, 1972) 1–81.
[5] J.A. Goguen, J.W. Thatcher and E.G. Wagner, Initial algebra approach to the specification, correctness, and implementation of abstract data types, in: R.T. Yeh, Ed., *Current Trends in Programming Methodology, Vol. IV, Data Structuring* (Prentice-Hall, Englewood Cliffs, NJ, 1978).
[6] I.P. Goldstein and D.G. Bobrow, A layered approach to software design, Technical Report CSL-80-5, Xerox Palo Alto Research Center (1980).
[7] J.V. Guttag and J.J. Horning, Formal specification as a design tool, *Proc. ACM Conference on Principles of Programming Languages*, Las Vegas, NV (1980) 251–261.
[8] K.L. Heninger, Specifying software requirements for complex systems: New techniques and their application, *Proc. IEEE Conference on Specifications of Reliable Software*, Boston, MA (1979) 1–13.
[9] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (8) (1978) 666–677.
[10] M.A. Jackson, *Principles of Program Design* (Academic Press, London, 1975).
[11] C.B. Jones, *Software Development: A Rigorous Approach* (Prentice-Hall, Englewood Cliffs, NJ, 1980).
[12] D. Kapur, Towards a theory for abstract data types, Technical Report 237, MIT Laboratory for Computer Science, Cambridge, MA (1980).
[13] J.G. Mitchell, W. Maybury and R. Sweet, Mesa language manual, Technical Report CSL-79-3, Xerox Palo Alto Research Center (1979).
[14] D.R. Musser, Abstract data type specification in the Affirm system, *IEEE Trans. Software Engrg.* **6** (1) (1980) 24–32.
[15] G.J. Myers, *Reliable Software Through Composite Design* (Petrocelli/Charter, New York, 1975).
[16] R. Nakajima, M. Honda and H. Nakahara, Hierarchical program specification and verification—A many-sorted logical approach, *Acta Informat.* **14** (1980) 135–155.
[17] D.L. Parnas, On the criteria to be used in decomposing systems into modules, *Comm. ACM* **15** (12) (1972) 1053–1058.
[18] E. Schmidt, Controlling large software development in a distributed environment, Ph.D. Thesis, U.C. Berkeley (1982) forthcoming.
[19] D. Scott, Toward a mathematical semantics for computer languages, in: J. Fox, Ed., *Proc. Symposium on Computers and Automata* (Polytechnic Institute of Brooklyn Press, New York, 1971).
[20] B. Sufrin, Formal specification of a display-oriented text editor, *Sci. Comput. Programming* **1** (3) (1982) 157–202.
[21] W.M. Turski, Design of large programs, IInf UW Reports Nr 97, Institute of Informatics, University of Warsaw (1980).
[22] P. Wegner, The Vienna definition language, *Comput. Surveys* **4** (1) (1972) 5–63.

[23] J.M. Wing, Bridging algebraic specifications and their implementations via interfaces, Ph.D. Thesis Proposal, MIT Laboratory for Computer Science, Cambridge, MA (1982).
[24] N. Wirth, Program development by stepwise refinement, *Comm. ACM* **14** (4) (1971) 221–227.
[25] J.L. Zachary, A syntax-directed specification editor, S.M. Thesis, MIT (1982) forthcoming.
[26] S.N. Zilles, Abstract specifications for data types, Technical Report 119, MIT Computation Structures Group (1974).