APPROACHES TO EXECUTABLE SPECIFICATIONS

Stephen W. Smoliar
Schlumberger-Doll Research
Old Quarry Road
P. O. Box 307
Ridgefield, Connecticut 06877

1 MOTIVATION

At the final panel discussion of the 5th International Conference on Software Engineering, the following observations were offered by Robert Balzer (summarized by Peter Neumann):[14]

> The specification must be the basis for programming. All modifications must begin there, with the program being rederived therefrom. Thus the specification becomes a rudimentary program, and all transformations are from one program form to another.
>
> If specifications are executable, then the original specification and all derived versions are programs, and all transforms are from programs to programs. Further, the executable specification can and should be used as the prototype. A prototype is then simply an incomplete implementation.

There is a suggestion of oxymoronic incongruity in the term "executable specification." Much of the pioneering work in software specification would invoke the metaphor of the blueprint or similar engineering drawing, but "hard" engineers certainly never expected these diagramatic abstractions to exhibit any form of behavior. Why should software specifications be different from other engineering specifications?

This question just begs to be answered with the obvious question: Why not? As software projects get more complex, the need for rapidly available prototypes becomes more essential. The obvious source for such a prototype lies in the specification, a statement which should constitute a formal agreement between the party who desires the software and the party who undertakes to provide it. If that specification can be formulated in a language which has operational semantics, then, de facto, the specification becomes the prototype; and the behavior of that prototype may be scrutinized to determine if it is, in fact, the behavior of the desired software product.

What constitutes operational semantics for a specification language? This is essentially the key question of this paper, which will consider a variety of approaches to specification languages, each of which supports an operational interpretation. One may proceed according to the following rule of thumb: A specification should be expressed in a notation which serves as a valid language for some (probably virtual) processor. One may then invoke rules which transform processor languages into processor languages. Thus, the specification will go through a series of stages of articulation as programs for a sequence of (again probably virtual) processors. The goal of this transformational undertaking is to end the sequence with a program for the target processor (the host for the software being developed).

This transformational approach to software development is by no means new. A fair amount of research in this area has already been undertaken by Robert Balzer.[2] Balzer has taken a state-oriented approach to specification: a program is specified in terms of a goal state, which may be expressed in terms of constraints on the properties of elements of the state. These constraints serve to determine valid states and state transitions. These state transitions serve to transform the static description of the goal state into a dynamic description of a program based on some set of operational primitives. (The primitives Balzer has worked with are those of a simple sequential backtracking language.)

This approach is similar to that taken by the CIP (Computer-aided, Intuition-guided Programming) project at the Technical University of Munich.[4] A key element of this project is the development of a general-purpose abstract language:

> This language must incorporate a variety of concepts, yet still retain a manageable size. It covers coherently the entire spectrum from problem specification tools to machine-oriented languages; it comprises such constructs as e.g., descriptive expressions and choices, predicates and quantification (used for abstract specifications and mode restrictions), recursive modes, recursive functions and non-deterministic conditionals (for the applicative formulation of algorithms), and variables, collective assignments, procedures, iteration, etc. (for the development towards machine language). According to different collections of constructs used in the formulation of a program, several particular languages "styles" (instead of different languages) can easily be distinguished.

The transformational aspect of this approach is involved with automating the transition between these different styles. Thus, one may begin with a specification articulated in the style of a purely descriptive expression and invoke a sequence of transformations which will ultimately represent the program in the style of a machine-oriented language.

A somewhat different approach is that provided by the specification language OBJ-T.[10] OBJ is a formal language for the expression of algebraic specifications; that is, the specifications are described in terms of sets (sorts), operations over those sets, and invariant equations which characterize the properties of the operations. OBJ is also an applicative, non-procedural programming language. OBJ-T provides an implementation of OBJ. The implementation is again a transformational one, now defined in terms of rewrite rules based on the equations which comprise the specification. Thus, an execution consists of tracing the implications of operations applied to some given sort object, initially represented as a closed expression.

In general, it would appear that expression-oriented representations are more conducive to transformation than are machine-oriented representations. This is because an expression-oriented representation is generally a static embodiment of dynamic behavior, as opposed to a prescription for a sequence of state transitions. This paper will survey a variety of expression-oriented representations which may serve as executable specifications. These representations include the lambda calculus, expressions in the functional style of John Backus,[1] the constructs of data flow languages, and algebraic approaches. This survey will be preceded by a brief digression on similarities between hardware specifications and software specifications.

2 SOFTWARE AND HARDWARE

The examples cited in Section 1 were concerned strictly with the specification of software. However, the current levels of complexity which arise in computer architecture make it obvious that specifications are also necessary for hardware. Here there is more of a tendency to think in terms of the blueprint metaphor. However, between the potential increase of functional capabilities afforded by VLSI and the control capabilities provided by microprogramming, it is no longer possible for hardware design to circumvent certain levels of description which are essentially algorithmic in nature. Hardware design is much more than the development of the appropriate circuit topology. It is the realization of a functional behavior in terms of some assumed set of functional primitives, and the circuit topology is simply the specification of how those primitives interact.

A variety of computer hardware description languages (CHDLs) have led a "double life" as simulation languages.[3] A specification encoded in one of these languages serves to drive a hardware simulation system, and the behavior of that simulation then serves as the operational interpretation of the specification. Unfortunately, most of these languages are structured as register transfer languages. The register transfer statement specifies how a given register is loaded with a given value; its semantics are essentially those of the traditional assignment statement. For this reason, most CHDLs bear a greater resemblance to conventional programming languages, augmented to accommodate some sort of topological declaration, than they do to specification languages. Furthermore, by tying the interpretation of the hardware specification down to the behavior of a simulator, such languages tend to evade one of the more useful properties of software specification languages, the fact that the static structure of the specification embodies in some significant way the dynamic nature of the specified behavior.

There have been a few attempts to transcend the limitations of register transfer statements and arrive at a more powerful and general hardware specification language. One such attempt has pursued the algebraic approach of OBJ, interpreting the individual hardware registers as sorts, rather than program variables.[7] This approach has the advantage of readily accommodating a hierarchic decomposition of the hardware being specified, but it has the disadvantage of concealing the basic topological description of the circuit being described. This information is kept implicit in the function compositions which are expressed in the equational identities.

An approach which addresses topological description more directly is one in which the hardware is represented by an expression in a functional programming language (such as a Backus FP system[1]).[8] In this notation the use of functionals essentially embodies necessary topological information regarding the connectivity of the functions on which they operate. A more general discussion of the possible relations between such functionals and network connectivity has been provided by Elliott Organick.[15]

A final approach which is also in the spirit of applicative and functional programming involves hardware specification in terms of a data flow notation.[17] The use of data flow constructs for software specifications will be discussed in Section 5. At this point it is sufficient to note that the principles discussed in that Section are as applicable to hardware specifications as they are to software specifications.

3 LAMBDA CALCULUS

Dialects of the lambda calculus, such as LISP, have proved their worth as programming languages many times over. More recently, the use of such languages for purposes of specification has also yielded fruitful results.[16] From this point of view, the prime virtue of such languages is that their semantics are strictly concerned with the behavior of functions. These functions may ultimately be considered simply as mappings from some domain space to some range space. Such a form of expression is excellent for the articulation of what a system is to do without bringing in the details of how it is to do it, a characteristic which Balzer regards as being essential to a good specification.[2]

Such languages also have the advantage of the utmost structural simplicity. There is only a single control structure: the application of a function to its arguments. Generally, there is also only one data structure, some variation on the S-expressions of LISP which combines atomic elements into an embedded list structure. This general purpose list structure may then serve as a representation of specific data structures which must be specified. Data structuring information may essentially be captured by predicates which determine whether any given list structure has the properties of the desired data structure. These predicates then serve to control the conditional expressions which constitute the body of most function definitions. From a more general point of view, functions are defined in terms of their operations on different abstract syntactic entities; and these entities are realized concretely by those S-expressions which satisfy the predicate associated with that given abstract entity type.[5]

Just as S-expressions provide a simple foundation for the elaboration of more complex data structures, so may more complex control structures be elaborated in terms of functional application. The basic mechanism for this elaboration is composition, whereby the operands of a function arise as a result of other functional applications. This mechanism, combined with the fact that functions may, themselves, serve as parameters and values of functions, endows lambda calculus dialects with the full power of Turing computability. Thus, the one notation serves as both an abstraction of relationships among functional components and a full-fledged programming language.

The operational semantics under which a lambda calculus specification becomes executable are essentially the reduction rules of lambda conversion. From an abstract point of view, even the basic LISP interpreter is nothing more than an implementation of these reduction rules.[12] However, some of the more recent lambda calculus dialects have employed much more powerful implementations of these rules. The basic approach to reduction taken by the LISP interpreter is that the expressions of a function's operands are reduced to normal form before reduction of the application of that function is undertaken. This is known as applicative order reduction.[19] The interpreter for the SASL language, on the other hand, performs normal order reduction. In this case, attempts are made to apply the reduction rules to the "outermost" applications of an expression first; and the more embedded expressions are not reduced unless such reductions cannot be performed.

Of course, the Church-Rosser Theorem proves that if both these reduction strategies terminate in a normal form, then they will terminate in the same form.[5] However, there are cases in which normal order reduction will yield a normal form; but applicative order reduction will not. In particular, an interpreter which supports normal order reduction will also support the manipulation of infinite data structures.[19] Such a facility seems particularly appropriate for issues of specification. Real-time systems, for example, may be modeled as transducers which map the infinite streams of signals received at sensors into the infinite streams of commands dispatched to effectors.[16] From this point of view, the operational semantics of a lambda calculus language may actually be more powerful than those of the more conventional sequential programming languages.

4 FUNCTIONAL EXPRESSIONS

As was observed in Section 3, one of the powerful elements of the lambda calculus is the ability of functions to serve as parameters and values of other functions. The functional style of John Backus essentially pursues the power of this approach to the development of an expression-based language which differs significantly from the lambda calculus. The crux of this difference concerns the elimination of the need for lambda variables. This is achieved through a notation which draws a distinction between functions, which map data objects into data objects, and functional forms (also known as functionals or operators), which synthesize functions through the combination of functions and data objects. A language based on such a notation is called an FP system.[1]

From the point of view of specification, an FP system is a fulfillment of the primary virtue of the lambda calculus--a representation of system behavior strictly in terms of functions. If one regards the postulation of lambda variables as an initial commitment to storage allocations, then the need for that commitment has been eliminated. Furthermore, the generalization of the lambda calculus to the accommodation of infinite streams, cited in Section 3 as valuable to the specification of real-time systems, may also be applied to FP systems. FQL is a functional style language which accommodates such streams; and this aspect of it has been used in the specification of a real-time ballistic missile defense (BMD) construct.[18]

Like LISP, FQL is an interpreted language.[6] It also provides a flexible environment for function definition, not unlike the environment afforded by APL. Consequently, specifications written in FQL are readily executable; and unspecified detail may be accommodated by functions defined to serve as stubs.

5 DATA FLOW CONSTRUCTS

One of the major critical issues of specification concerns the need to represent concurrency. One of the reasons that Balzer emphasized the significance of the "what" over that of the "how"[2] was to avoid a specification which would impose constraints of sequential execution, particularly if the target environment consisted of multiple processing facilities. Both the lambda calculus and functional style languages tend to be essentially sequential in nature, although they may be interpreted in ways which may allow for parallelism.[9]

An alternative form of specification is one in which possibilities for parellelism are implicit in the notation. This is one of the

virtues claimed by data flow languages such as VAL.[13] Such languages serve to represent a computation as a network of functions. The network serves to define the communication of message packets among the component functions. These message packets provide the means for the propogation of input parameters to a function and output values yielded by a function. Because the components are functions, each one may, at least in theory, be assigned to a distinct processor; and the network defines how these processors must be interconnected.

The actual VAL notation, however, is much more like a programming language than a specification language. The "internal" description of an individual function is still given in essentially sequential constructs. Consequently, the language does not lend itself to any sort of abstract representation which may be filled out through hierarchical refinement. Thus, VAL is primarily concerned with the "how" of a system; however, this "how" is defined with sufficient generality to allow for a varity of implementations on multiple processing resources.

A notation which is more conducive to abstractions which may be subsequently "fleshed out" by stepwise refinement is the language developed by Gilles Kahn and David MacQueen.[11] This language provides a reconfiguration construct, by which a given node in a dataflow graph may be refined into a representation as a subgraph. All that reconfiguration demands is that those edges which were previously directed into and out of the node be suitably attached in the subgraph structure. This means that one may provide a specification in terms of one of these networks in which the individual function nodes are interpreted by stubs, and an implementation is achieved when one eventually arrives at a graph all of whose nodes are realized by executable code.

6 ALGEBRAIC REPRESENTATIONS

As was observed in Section 1, an algebraic specification involves a description in terms of sets, functions over those sets, and invariant equations over the functions. In many ways such a specification may be regarded as the ultimate embodiment of the "what." It provides a representation of those properties which must be satisfied without necessarily imposing any commitment as to how they be satisfied. The absence of such a commitment, however, would seem to imply that such a representation does not lend itself to execution.

The fallacy of this reasoning has been admirably dispatched by the OBJ-T specification language.[10] The basic approach to the execution of OBJ-T is the same as that of the lambda calculus: the invariant equations which comprise a specification may be interpreted as reduction rules. Thus, given an OBJ-T specification and an initial expression, that expression may be subject to a sequence of rewritings determined by the equations of the specification. The actual implementation of OBJ-T is endowed with a set of rules which determine how such equations may be invoked in a reduction operations. These rules insure that the rewriting process will eventually terminate and that the reduction will be as efficient as possible.[10] Thus, even the abstraction of an algebraic specification may lend itself to an operational interpretation.

7 CONCLUSIONS

Each of the four approaches considered in the preceding sections--lambda calculus, functional expressions, data flow constructs, and algebraic representations--should be viewed in terms of the limitation suggested by the title of this paper. An approach to specification does not necessarily constitute a satisfactory method. What is promising, however, is that all four of these approaches allow for some form of executable interpretation. As the complexity of software increases, it will become more and more desirable that a specification also serve as a prototype, in which case each of these four approaches should be considered for its ultimate applicability to specification in the software crises to come.

REFERENCES

1. J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Comm. ACM, Vol. 21, pp. 613-641, August 1978.

2. R. Balzer, "Transformational implementation: an example," unpublished report, USC Information Sciences Institute, August 1979.

3. M. R. Barbacci, "A comparison of register transfer languages for describing computers and digital systems," IEEE Transactions on Computers, Vol. C-24, pp. 137-150, February 1975.

4. F. L. Bauer, et al., "Towards a wide spectrum language to support program specification and program development," in F. L. Bauer and M. Broy, Program Construction, Springer-Verlag, Berlin, 1979, pp. 543-552.

5. W. H. Burge, Recursive Programming Techniques, Addison-Wesley, Reading, 1975.

6. R. E. Frankel, "FQL-the design and implementation of a functional database query language."

7. R. E. Frankel and S. W. Smoliar, "Beyond register transfer: an algebraic approach for architectural description," Proc. 4th International Symposium on Computer Hardware Description Languages, Palo Alto, California (October 1979), pp. 1-5.

8. R. E. Frankel and S. W. Smoliar, "Digital systems as mathematical expressions," Proc. COMPCON Spring 81, San Francisco, California (February 1981), pp. 414-416.

9. D. P. Friedman and D. S. Wise, "Aspects of applicative programming for parallel processing," IEEE Transactions on Computers, Vol. C-27, pp. 289-296, April 1978.

10. J. A. Goguen and J. J. Tardo, "An introduction to OBJ: a language for writing and testing formal algebraic program specifications," Proc. Specifications of Reliable Software, Cambridge, Massachusetts (April 1979), pp. 170-189.

11. G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," Information Processing 77 (IFIP proceedings), Toronto, Ontario (August 1977), pp. 993-998.

12. J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," Comm. ACM, Vol. 3, pp. 184-195, April 1960.

13. J. R. McGraw, "Data flow computing--software development," IEEE Transactions on Computers, Vol. C-29, pp. 1095-1103, December 1980.

14. P. G. Neumann and S. Gerhart, "Some reflections on the 5th ICSE," Software Engineering Notes, Vol. 6, pp. 5-7. April 1981.

15. E. I. Organick, "New directions in computer systems architecture," EURO MICRO Journal, Vol. 5, pp. 190-202, July 1979.

16. S. W. Smoliar, "Using applicative techniques to design distributed systems," Proc. Specifications of Reliable Software, Cambridge, Massachusetts (April 1979), pp. 150-161.

17. S. W. Smoliar, "Simulating distributed systems: a two-level approach," Proc. AIAA Computers in Aerospace III Conference, San Diego, California (October 1981), to appear.

18. S. W. Smoliar, "Applicative and functional programming," in C. R. Vick and C. V. Ramamoorthy, Handbook on Software Engineering, Van Nostrand Reinhold, to appear, pp. 11-1 - 11-55.

19. D. A. Turner, "A new implementation technique for applicative languages," Software - Practice and Experience, Vol. 9, pp. 31-49, January 1979.