INFORMATION SCIENCES **57–58**, 347–359 (1991)

The Role for Executable Specifications in System Maintenance*

MARVIN V. ZELKOWITZ

Inst. for Advanced Computer Studies, and Department of Computer Science, University of Maryland, College Park, Maryland 20742

and

SERGIO CARDENAS

Department of Computer Science, University of Maryland, College Park, Maryland 20742

ABSTRACT

As software becomes increasingly complex, two attributes of the system life cycle are taking on more important roles. We need the ability to formally specify the functionality of the systems we build in order to minimize costly development problems and, with long life times, we need the ability to enhance existing systems with new features in order to prolong their usefulness. This enhancement process also needs a mechanism for formally defining any new functionality on data objects placed upon the system. This paper describes the AS* research project which is addressing this issue. AS* is a language-independent specification language embedded within an existing programming language for the purpose of providing extensions to existing systems. This paper discusses the designs of AS*, the structure of the prototype implementation, and describes some early experiences using the system.

1. INTRODUCTION

As software becomes increasingly complex, two attributes of the system life cycle are taking on more important roles. We need the ability to formally specify the functionality of the systems we build in order to minimize costly development problems, and with long life times, we need the ability to enhance

* The AS* research project was developed as a joint project with Sergio Antoy, now at Portland State University, Portland OR, and Paola Forcheri and Maria Teresa Molfino of Instituto per la Matematica Applicata of Genoa, Italy. Research support at the University of Maryland was partially provided by Air Force Office of Scientific Research grants AFOSR-87-0130 and AFOSR-90-0031.

655 Avenue of the Americas, New York, NY 10010 0020-0255/91/$03.50

existing systems with new features in order to prolong their usefulness. The role of formal specifications (e.g., algebraic, axiomatic, functional, or operational specifications) is currently of great interest within the computer science research community as a means to address the first of these problems.

However, formal specifications are not created in a vacuum. The underlying system that was initially specified will evolve and change, and the need to make enhancements to the basic system (and hence change the underlying formal specification) will be an ongoing life cycle concern.

In enhancing a system with new features, the "established" research model requires a development group to refine the requirements, redo the specifications, modify the design and alter the code. But before a system is altered, what we usually have is a requirements document long unused, an out-of-date specification that might have reflected the original goals of the project, a design that is undoubtedly incomplete or incorrect and a source program that ultimately needs to be modified. Traceability of new source code feature back through design, specifications, and ultimately back through to the requirements is certainly a need, but we have no effective mechanism to achieve that today. However, in order to enhance a system, in the midst of all this apparent confusion, we do have one clear advantage over the original designers of the system—we have a running source program that can provide the basis of an "oracle" for testing new enhancements.

We have been studying this problem within the context of an executable specifications language which we have named AS* (for Algebraic Specifications extended). The goal is to design such a specifications language that: (1) has the formal properties needed to correctly specify system functionality; (2) can be compiled into executable programs; and (3) can be added to existing systems in order to provide for system enhancements. By developing such a language, we see its introduction into the software life cycle as follows:

(1) A new feature is modeled (i.e., specified) by the specifications language.

(2) The feature is automatically tested (i.e., prototyped) in the context of the existing system for appropriate behavior.

(3) The feature is designed in the native programming language in a more efficient manner.

(4) If necessary, parts of this new implementation are redesigned to further increase the efficiency of the new system.

(5) At any time we can execute our correct model (i.e., step (2) prototype) in order to ensure that each successive change or enhancement doesn't change the desired functionality.

This approach differs from others [9, 10] in that the specifications are executed within the context of a running system in order to test its behavior.

Thus, while other models can simulate the behavior of an entire system as part of a system design activity, they are at a loss at simulating small extensions within a larger existing product—the essence of the important maintenance aspect of the life cycle.

In Section 2 of this paper we describe our AS* specification language and we describe our initial implementation of a Pascal environment containing AS* specifications (ASPascal). Section 3 gives some initial experiences in using our prototype development tool.

2. AS*

Software design consists of the creation of complex data objects, usually referred to as abstract data types, and the definition of functions that operate on these abstract objects. Several models for specifying programs have been developed including inductive assertions [3, 6] and algebraic specifications [4, 5]. We are using algebraic specification technology consisting of a series of axioms or equations relating the operations of the abstract type to each other, as our basic model.

The Knuth Bendix algorithm [8] applied to these specifications defines a proof of adequacy by showing the equivalence of supposedly equal terms to the same constant terms. Since the Knuth Bendix algorithm uses an ordering transformation that converts one term to a "simpler" term, the algorithm defines an operation that can be "executed" and proven to terminate. Therefore, any set of axioms that is "Knuth Bendix" can be transformed mechanically into a series of transformations that can be executed. It is this transformation that is the basis for our executable specifications.

2.1. AS SYNTAX*

Similar to the initial algebra approach of other term-rewriting systems [2, 7], an AS* specification contains three features: (1) a *sort* name which defines a new abstract object and its *constructors*, functions to build objects of that sort; (2) a *signature* which defines the functionality of set of *defined operations* and constructors for manipulating the abstract sort objects; and (3) a set of *equations* (or *axioms*) which inter-relate the semantics of the defined operations and constructors.

Specifications can be *generic* or *explicit*. A generic specification is a schema that resides (usually) in a program library and contains parameters (variables, operations, and other sorts) that are instantiated when the specification is used in an actual program. An explicit specification is a refinement of a generic specification that substitutes actual arguments for the specification parameters.

```
(1)  sort sequence [sort something] is
(2)  constructor
(3)    epsilon;
(4)    cons : something, sequence;
(5)  operation head : sequence → something is axiom
(6)    head(epsilon) == ?;
(7)    head(cons(X,Y)) == X;
(8)    end;
```

Fig. 1. Example of sequence specification

Figure 1 gives a simple example of a specification for a sequence. Line (1) specifies that we are defining a class of objects of sort *sequence*, and indicates that the new object will require an internal sort *something* that will be specified in a later binding. Lines (2)–(4) define the two constructors needed to create an object of this sort: *epsilon* to return the empty object of sort *sequence* and *cons* which takes an element and a sequence and returns a new sequence with the element in it. The functionality of each constructor is given after its name with the sort name *sequence* implied as last (e.g., *epsilon* returns an empty *sequence* and *cons* requires a *something* and a *sequence* and returns a *sequence*). *Epsilon* initializes objects of this sort and *cons* creates complex objects.

This object is manipulated by means of a set of *defined operations*, which only *head* is given with its *signature* on line (5). It is defined by the rewrite rules (axioms) on lines (6)–(7) which say to return the last element included into the sequence by the *cons* function. Each axiom consists of a rewrite rule where the expression on the left hand side of the equality operator == consists of a functional value of free variables and the right hand side expression gives its meaning via an arbitrary expression involving the left hand side variables.

The '?' on line (6) is equivalent to an abnormally terminating computation. Our implementation stops execution and issues an error message when this occurs. Within a program, specifications appear as function calls in the host programming language to the various operations and constructors defined in a sort.

Much like Larch and Larch/CLU, AS* specifications are independent of the underlying programming language and must be defined relative to any concrete language. Libraries of generic specifications can be used to form the basis of a re-use methodology where the generic specification is refined to an explicit specification in a specific programming language by binding the generic sorts to specific programming language types. In our case we use Pascal

as our implementation vehicle, so to create ASPascal, the extension to Pascal that contains AS* specifications, we indicate a linkage between a Pascal object and an AS* *sort*.

An *explicit specification* is created by a refinement of a generic specification via the **use** clause. Syntactically, a refinement of a specification is:

sort *identifier* **is use** *sortname [parameterlist]* **end**

where *identifier* is the refined sort name, *sortname* is the sort schema to be refined, and *parameterlist* is the list of actual functions and sorts that are substituted for the parameters in the sort definition. All of the defined operations in the original generic sort are now redefined in the context of this new refined sort.

The AS* specification:

sort *intsequence* **is use** *sequence [integer]* **end**;

refines the generic sort *sequence* and indicates that a new sort *intsequence* is created by modifying *sequence* with a binding of Pascal integers to the free sort *something* of Figure 1.

Sorts are linked into Pascal by interpreting a specification like

sort *newsort* **is** . . .

as equivalent to the Pascal type declaration

type *newsort* = . . .

The primitive Pascal scalar types (char, boolean, integer, real) may all be used in abstract sort definitions, and any explicit sort may also be used in a refinement. Thus,

var *A: intsequence*;

simply creates a Pascal variable A which is of (sort) type *intsequence*.

By using alternative bindings, we greatly expand reuse of specifications. For example, real sequences could be created as:

sort *realsequence* **is use** *sequence [real]* **end;**

Similarly, a sort such as a *book* (which is unspecified in this paper, but could either be a type definition in the host language consisting of a record of author,

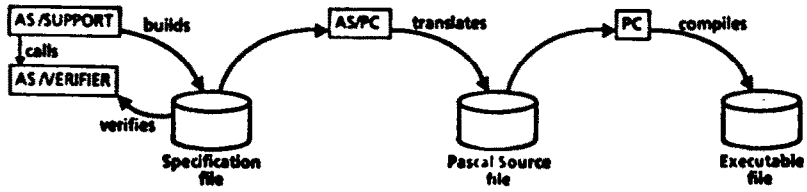

Fig. 2. AS* toolset

title, publisher, . . . , or could be another sort specification with similar attributes) could be used to create a type *library* as:

sort *library* **is use** *sequence [book]* **end;**

2.2. AS ENVIRONMENT*

Considering an algebraic specification as a term rewriting system is fairly straightforward; however, correctly specifying such algebraic axioms is not easy. Therefore, without computer-based tools to aid in the process, the task is extremely difficult. We have therefore constructed a series of integrated tools that both do the conversion from axiomatic definition into executable source program, and also provide aid in validating the specifications. A prototype implementation of the AS* system has been constructed and executes on the SUN 3 workstation under Berkeley UNIX 4.3. Figure 2 represents the initial AS* environment that has been constructed.

The four components are:

(1) AS/SUPPORT is a modification to the SUPPORT environment [11] which provides text-editing capabilities for creating specifications. SUPPORT is an integrated environment based upon a syntax-directed editor built to create, execute, and test programs written in a subset of Pascal. Since the language it processes is determined by an externally defined grammar file which also includes semantic information and screen unparsing commands, it was easy to modify the grammar to process Pascal as extended by the AS* syntax.

(2) AS/VERIFIER, a Prolog program, is called by AS/SUPPORT and verifies the set of axioms. AS/SUPPORT converts the sort axioms into a series of Horn clauses suitable for analysis by Prolog and then calls AS/VERIFIER, which then checks the convergence of these clauses relative to the Knuth Bendix algorithm. Details of the verification algorithm are given elsewhere [1]. If any error is found, an appropriate message is relayed back to AS/SUPPORT and displayed to the user.

Other verifiers usually interact with a user who manually indicates approval to continue the process or terminate. In our case, AS/VERIFIER does a single pass over the axioms and then terminates. AS/SUPPORT will reinvoke AS/VERIFIER if the user modifies the sort definition. Thus the system appears interactive to the user, but the underlying model is not.

(3) AS/PC is the translator written using YACC that converts specifications into standard Pascal source programs. Each axiom is implemented as an **if** statement that checks on the validity of the left-hand-side arguments, and, if true, replaces the value by the right-hand-side expression. Since AS/VERIFIER has already proven that the axiom is "Knuth Bendix," the right-hand-side expression is simpler than the left and the process will eventually terminate at some constant terms.

(4) PC is any standard Pascal compiler. At this point, the specifications have been converted to standard Pascal, and any comparable compiler can be used for compilation and execution.

The basic SUPPORT environment has been in use since 1986, and the AS* extension, ASPascal, has been operational since the fall of 1989. More complete details of its implementation are described elsewhere [1].

3. USING AS*

The initial implementation of AS* has been designed as a mechanism to enhance existing Pascal source programs. Figures 3 through 6 represent a simple example of how such a system could be used.

In Figure 3, a simple circuit of *and* and *or* gates is specified using the AS* formalism. The sort *switch* is defined to represent one type of relevant gate with constructors *andg* and *org* representing primitive gates and the operation *SwType* returning the gate type of its argument.

The sort *circuit* consists of primitive wires (constructors *high* and *low* standing for high and low voltages which represent the conditions *true* and *false*), a constructor *build* that constructs complex circuits out of simpler circuits, and one operation *print* that determines the output characteristics of a circuit. Operations *max* and *min* are specified to be local functions used within the sort *circuit*.

The main body of this sample program simply builds a binary circuit of $(2^{18} - 1)$ primitive gates and then computes the output value of the resulting circuit.

Figure 4 represents a first approximation of the new feature in standard Pascal. It was created by designing a source program based upon the initial sort prototype of Figure 3. It should be clear from Figures 3 and 4 that the algebraic model of specification is closely related to an applicative (i.e., func-

```
program main (input,output);
   sort switch is
        constructor
           andg; org;
        operation SwType: switch -> integer is
           axiom
           SwType(andg) == 1;
           SwType(org) == 2;
         end;
   end;

   sort circuit is
      constructor
         low; high;                              {primitive wires}
         build: switch, circuit, circuit;  {complex circuits}

      operation min: integer, integer -> integer is
         axiom
         min(A,B) == if A<B then A else B;
         end;

      operation max: integer, integer -> integer is
         axiom
         max(A,B) == if A>B then A else B;
         end;

      operation print: circuit -> integer is
         axiom
         print(low) == 0;
         print(high) == 5;
         print(build(T, A, B)) ==
                if (SwType(T)=SwType(andg))
                       then min(print(A),print(B))
                       else max(print(A),print(B));
         end;
   end;

var
  gateX : circuit;
  i:integer;

begin {Create a basic gate with two inputs}
  gateX := build(org, high, low);
  writeln('HIGH or LOW',print(gateX));
  i:=0;          {create a binary tree of gates}
  while i<17 do
        begin
          gateX := build(andg, gateX, gateX);
        i:=i+1;
        end;
  writeln('DONE',print(gateX));
end.
```

Fig. 3. Sample specification

```
program main (input,output);
        type gatetype=(andg, org, high, low);
        circuit = ^gate;
        gate= record
                circuit: gatetype;
                wire1: circuit;
                wire2: circuit
                end;
        function getcircuit(circuittype:gatetype): circuit;
                var x: circuit;
                begin
                new(x);
                x^.circuit := circuittype;
                getcircuit := x
                end;
        function build(circuittype:gatetype; A, B: circuit):circuit;
                var x:circuit;
                begin
                x:= getcircuit(circuittype);
                x^.wire1 := A;
                x^.wire2 := B;
                build := x
                end;
        function min(A,B:integer): integer;
                begin
                if A<B then min:=A else min:=B
                end;
        function max(A,B:integer): integer;
                begin
                if A>B then max:=A else max:=B
                end;
        function print(x:circuit):integer;
                begin
                case x^.circuit of
                high: print:= 5;
                low: print:=0;
                andg: print:= min(print(x^.wire1),print(x^.wire2));
                org: print:= max(print(x^.wire1),print(x^.wire2))
                end {case}
                end; {print}
var
  gateX : circuit;
  i: integer;
  highgate, lowgate: circuit;
begin {Create a basic gate with two inputs}
  highgate := getcircuit(high);
  lowgate := getcircuit(low);
  gateX := build(org, highgate, lowgate);
  writeln('HIGH or LOW',print(gateX));
  i:=0;         {create a binary tree of gates}
  while i<17 do
        begin
        gateX := build(andg, gateX, gateX);
        i:=i+1;
        end;
  writeln('DONE',print(gateX));
end.
```

Fig. 4. Initial Pascal version of sample specification

```
function circuit_print(arg1: circuit): integer;
begin
  if arg1^.tag = circuit_build_tag then
     if (switch_SwType(arg1^.circuit_build_arg1) = switch_SwType(switch_andg))
        then circuit_print := circuit_min(circuit_print
        (arg1^.circuit_build_arg2), circuit_print(arg1^.circuit_build_arg3))
        else circuit_print := circuit_max(circuit_print
        (arg1^.circuit_build_arg2), circuit_print(arg1^.circuit_build_arg3))
     else if arg1^.tag = circuit_high_tag then
        circuit_print := 5
     else if arg1^.tag = circuit_low_tag then
        circuit_print := 0
     else circuit_ECall('circuit_print undefined for argument')
end;
```

Fig. 5. Translation by AS/PC of function print

tional) style of programming. Operations *min*, *max* and *print* are almost direct copies of their formal definitions. The AS* model, however, greatly simplifies storage management. While simply defining *andg*, *org*, *high*, and *low* as constructors allows for AS* to automatically build and allocate correct data structures, the Pascal equivalent required careful deliberate design steps in order to implement correctly.

In developing this initial implementation, only one change had to be made to the body of the Pascal program. We had to add variables for primitive true and false gates (e.g., *highgate* and *lowgate*) to compensate for the automatic generation of such gates with the formal specification of constructors *andg* and *org*. No other changes were made to the Pascal program.

For those curious, Figure 5 is an example of the output from AS/PC, and is the source program for function *print*. As Figure 5 demonstrates, AS/PC translates operation *print* defined in sort *circuit* into the procedure name *circuit_print*. This code is highly structured and is not meant to be read by most programmers. The user is expected to develop axioms via AS/SUPPORT and AS/VERIFIER and then execute the resulting program. Translation of the specifications into Pascal should occur directly from the sort definitions. Knowledge of the resulting Pascal output from AS/PC should not be crucial for verification of the program.

The formal specification for *print* was kept simple by using a recursive algorithm that required for computing the value of a circuit, first computing the values for the components of that circuit. Using the fact that in *and* gates, the result is *low* if the first input is low and in *or* gates the result is *high* if the first input is high, regardless of the second input, Figure 6 represents an easy optimization to the first Pascal implementation.

```
function print(x:circuit):integer;
        var cirvalue:integer;
        begin
        case x^.circuit of
        high: print:= 5;
        low: print:=0;
        andg: begin
            cirvalue := print(x^.wire1);
            if cirvalue=0 then print:= 0
                else print:=print(x^.wire2)
            end;
        org: begin
            cirvalue := print(x^.wire1);
            if cirvalue=5 then print:= 5
                else print:=print(x^.wire2)
            end
        end {case}
end; {print}
```

Fig. 6. Optimized print function

All three versions had extremely different execution times on a SUN 3/60 workstation, as given by the following table:

Prototype	27.9 sec
Initial Pascal	5.4 sec
Optimized Pascal	3.3 sec

The initial Pascall program represents about a fivefold improvement over the initial prototype and the simple optimized version represents almost another factor-of-two improvement for this simple example. This process of local optimization can be continued for further refinement of the program.

4. CONCLUSIONS

AS* is a specification language which has been designed to aid in system enhancement as a prototyping and code re-use tool within the context of extending existing source code systems. Language-independent specifications are defined via a term-rewriting notation and a Knuth Bendix verifier checks for their termination and executability. A prototype implementation processes ASPascal, a Pascal refinement to the AS* specification language.

The automatic execution of an AS* specification is obviously inefficient; however, as a prototyping tool, quick execution is not of primary importance.

The previous example shows that improvement of perhaps a factor of 10 can be achieved by locally improving the source program. In system maintenance, due to the complexity of the underlying system, it might be preferable to give up a few percentage points of execution time for ease in understanding and building the desired extensions. Of obvious interest and needing further study is this tradeoff between ease of use and performance penalty.

From our experience to date, development of the axioms requires an initial period of adjustment (which would be relatively easier for a designer familiar in an applicative language like LISP), but is not considered difficult. Storage management is considerably simpler with these axioms. However, the important point is that these axioms provide a much greater degree of control over functionality and verification of the underlying process of system enhancement—the crucial aspect in system maintenance.

This system addresses many of the important specifications and code reuse problems that are of interest today. Specifications are formal yet executable, and can easily be mapped into a variety of programming languages. Maintenance is enhanced via tests on the existing source program. Determining properties of specific abstract data type implementations is explicit and should help in the process of understanding and re-using source program libraries.

REFERENCES

1. S. Antoy, P. Forcheri, M. T. Molfino, M. V. Zelkowitz, Rapid prototyping of system enhancements, *Proc. of 1st Int. Conf. on System Integration*, Morristown, NJ, 1990.
2. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*, Springer-Verlag, Berlin Heidelberg, Germany, 1985.
3. R. W. Floyd, Assigning Meanings to Programs, *Proc. Symp. Appl. Math.* 19:19–32 (1967).
4. J. A. Goguen, J. W. Thatcher, E. G. Wagner, An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in *Current Trends in Programming Methodology* (R. T. Yeh, Ed.), Prentice-Hall, Englewood Cliff, NJ, 1978, pp. 80–149.
5. J. V. Guttag and J. J. Horning, The algebraic specifications of abstract data types, *Acta Informatica* 10:27–62 (1978).
6. C. A. R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM* 12:576–583 (1969).
7. G. Huet and D. Oppen, Equations and rewrite rules: A survey, in *Formal Language Theory* (R. Book, Ed.), Academic Press, 1980, pp. 349–405.
8. D. Knuth and P. Bendix, Simple word problems in universal algebras, in *Computational Problems in Abstract Algebra* (J. Leech, Ed.), Pergamon Press, NY, 1970, pp. 263–297.
9. J. M. Wing, Writing Larch interface language specifications, *ACM Transactions on Programming Languages and Systems* 9:1–24 (1987).

10. P. Zave, An operational approach to requirements specification for embedded systems, *IEEE Transactions on Software Engineering* 8:250–269 (1982).
11. M. V. Zelkowitz, B. Kowalchack, D. Itkin, L. Herman, Experiences building a syntax directed editor, *Software Engineering Journal* 4:294–300 (1989).

Received 7 February 1990; revised 25 July 1990