1983

Interface Issues in a Software Parts Technology

John R. Rice
Purdue University, jrr@cs.purdue.edu

Herbert D. Schwetman

Report Number:
83-447

To appear in:
Reusability in Software
E. Biggerstaff, editor, 1983

TR 447

INTERFACE ISSUES IN A SOFTWARE PARTS TECHNOLOGY

John R. Rice
Herbert D. Schwetman

Purdue University

ABSTRACT

A mature software parts technology will include tens of thousands of software parts available in a common environment. In principle, a programmer can attempt to combine any two available parts so the technology must provide robust mechanisms to insure reliable and meaningful parts composition. Existing approaches are briefly surveyed and we note that the principal mechanism in current use is syntatic (or type) checking which may occur at compile-time (Pascal), load-time (Ada) and run-time (Intel 432). There must also be a subsequent semantic checking which occurs at both load-time and run time. The PROTRAN system is an instance where run-time semantic checks of certain types are systematically used.

We describe an interface specification structure for all checking required for the highly reliable use of software parts. We identify three levels of interfaces and associated syntatic and semantic checking: global, part-specific and problem-specific. At the global level, most of the checking is syntatic. For specific parts not only must the data be of the correct type, but it must be valid for the part; e.g., a parameter is not only an integer, but a valid index into a particular array. Problem specific checking means that the data is appropriate to the problem; e.g. not only a matrix, but a matrix with certain properties. The essential role of standards and conventions is discussed, and an assessment of the trade-offs is made.

1. INTRODUCTION

In a recent article, Wasserman and Gutz (1982) present their views on "The Future of Programming". In the section on prospects for the medium term, they present as one of four expected changes: "development of certified software components - a body of rigorously tested and thoroughly documented software modules will be created and available for easy incorporation in new systems". More recently, the Department of Defense STARS program has focused on "reusable software parts" [Batz et al, 1983] as one of the major approaches to improving software productivity. One of the major technical issues in software parts is how to define the interfaces between parts, this is sometimes called the semantic interface problem. It is this problem that we address here.

We have been engaged in discussions since 1980 concerning the development of a software parts technology [Comer et al, 1980]. The comments of Doug Comer, Larry Snyder and Peter Denning influenced our views considerably. While we acknowledge their contributions, they bear no responsibility for the particular material presented here.

We first introduce some background about software parts and the programming environment which will be required to support a software parts technology. We then define the semantic interface problem and present our solution of it.

1.1. Background

The need for sharing and reusing code has been known for many years. One of the earliest attempts at reusing code was the subroutine library. For example, the SHARE Program Library is a repository of subroutines donated by users of large IBM systems. Prospective users can obtain copies of selected routines and incorporate them into their own software. It is our opinion that, while many of these routines are quite useful, the number of routines which fail to work as desired means that this library is an unreliable source of software. Software which is volunteered is susceptible to being unreliable and hence a collection of such software is almost useless.

A much more successful subroutine library is the one from IMSL, Inc. which is in the business of selling a library of subroutines for common mathematical and statistical procedures. Each routine is listed and described in the Reference

Manual [IMSL, 1979] and its quality is assured by the seller. Because the company is "in the business" of providing software, they have become proficient in the construction, distribution and maintenance of these software components. Today, a programmer operating in a computing center which makes the IMSL library available is literally wasting time and money if he resorts to writing software which performs any of the functions supplied by IMSL. It is much more efficient to locate and use the linear equation solver from IMSL than to write one from scratch. The subroutine library from NAG Inc. is of similar scope and quality.

The UNIX operating system [UNIX, 1978] (UNIX is copyrighted by Bell Telephone Laboratories) supports parts based computing in a manner which is different from subroutine libraries. In UNIX, the user has access to a large number of programs, each of which performs a simple function. By using the pipe mechanism (see below), these programs can be "assembled" into larger commands which perform more complex sets of functions. Each program is written so as to take its input from the "standard output file". Two programs can be invoked and be connected by a "pipe"; this means that the standard output of the first program is the standard input of the second. Thus, sequences of programs can be connected together by linear streams of characters. The large number of programs available on UNIX and the pipe mechanism mean that UNIX programmers are able to operate in a environment in which software parts are used as building blocks. The UNIX Programmer's Manual describes each part in terms of its function, inputs, outputs and error conditions. Since the source form for every program in UNIX is on-line, existing parts can be tailored to meet a need, if necessary. This is usually much less costly than starting from scratch.

There are other examples. APL programmers have access to sets of "idioms". These are predefined functions which perform certain desired tasks. A user can invoke an idiom, to perform a needed function, as opposed to writing a new one.

Many people are trying to distribute sets of software parts for micro processors. One example of this is Scientific Enterprises, Inc., with a product called XM-80 Software Components. This is a set of macros for use with the Macro-80 relocatable assembler for the Z80 microprocessor (Macro-80 is a product of Microsoft, Inc.). The XM-80 set of routines allows the programmer to invoke macros and subroutines to perform many commonly needed functions. Each macro is described with a data sheet [XM80, 1980].

The STARS effort [DoD, 1983] intends to develop large sets of software parts in the form of Ada packages. Packages are to be developed for a variety of application areas and the hope is not only to reuse software, but also to provide a "lingua franca" for the practitioners in various disciplines and subdisciplines (outside computer science).

1.2. Goals

The goal of a software parts technology is the development of a programming environment in which reusing code is the norm, not the exception. We feel strongly that a programmer, when faced with a programming task, should start looking for existing software parts instead of starting to write.

Consider the analogy with the process routinely used in the design and fabrication of digital electronic components. The steps followed seem to include:

- obtain specifications: these could include the function of the component in terms of its inputs and outputs, the required speed, size, power consumption and logic type,
- search catalogs: catalogs of components, usually integrated circuits or chips, are searched until the right set of parts can be located,
- order parts,
- design interconnections,
- build bench model, test and refine,
- obtain printed circuit boards or wire wrap boards,
- fabricate in pilot quantities,
- redesign: reduce costs and improve performance by using customized components for high volume items.

The key observation in this analogy is that it is only in the case of a high volume item (or when performance extremes are needed) that the designer would consider fabricating new components. The designer is not building chips and is not operating at the level of discrete components. The parts are at a higher functional level. This analogy is explored in more detail in Project Quanta [Comer et al, 1980] and [Wasserman and Gutz, 1982].

1.3. Requirements

In order for a software parts technology to become useful, there are at least three features which have to be

present; these include:

1. A large supply of useful, reliable parts,

2. A catalog of parts, making them easy to locate and evaluate, and

3. A mechanism for connecting parts together, so as to form more complex objects.

There must also be economic justification; if programming based on software parts "catches on", it will be because it makes sense economically.

We can evaluate IMSL and UNIX with respect to these requirements:

IMSL - The library of IMSL functions certainly provides a large supply of reliable parts, useful within the area of mathematical and statistical applications,

- The IMSL documentation lists and describes each routine,

- The interconnection scheme is not part of IMSL; interconnection is accomplished by writing a main (or driver) program and it associated data areas.

UNIX - The UNIX library of commands and programs constitutes a fairly extensive set of parts; these meet a large variety of needs in several areas, including text handling, data handling, and interactive inquiry.

- The UNIX Programmer's Manual is a comprehensive list of all programs; at Purdue, we have also prepared a KWIC index of key terms,

- The pipe mechanism described above serves as the interconnection scheme; the major limitation of the pipe is that it is limited to transmitting a single stream of characters.

2. INTERFACES: EXISTING AND POTENTIAL

2.1. Framework

The background discussion mentions a number of nascent software parts environments and their related interfaces. These interfaces can be classified by when the interface is checked and what checking is made; this is shown in Table 1. The entries in the table are examples of existing systems that do interface checking at the indicated point.

Table 1. The times and types of interface checks with examples:

When done	Type of checking: Syntax	Type of checking: Semantic
Compile-time	Pascal	-
Load-time	Ada	-
Run-time	Intel 432	Protran

Semantic checking at compile time is difficult because it would require all code and data objects to be present then. Furthermore, certain semantic checks cannot be made in advance of executing or pseudo-executing the program. Thus we believe that very little semantic checking will be done at compile time.

More semantic checking is feasible at load time, all the code is present and one could check that the arguments to all procedures form "well-posed" computations. Still, this checking is necessarily incomplete and requires additional facilities in the loader. Indeed, it is not clear that such checking would be more than sophisticated syntax checking. That is, one not only verifies that procedure arguments are individually of the correct type, but that the combination of types and attributes satisfy certain constraints.

Note that, with detailed checking and many data attributes, it is infeasible to have strong type checking in the sense that a single procedure accepts only a single combination of types and attributes. Visualize a matrix multiply procedure with type of elements (real, integer, etc.), precision of elements, precision of product, row-size and column-size. It would require 400,000 distinct procedures just to handle real matrices of size 20 by 20 or less and digits of precision of 10 or less. However, it is feasible to check at load time if the product precision is less than equal to the two input precisions.

Space precludes a detailed discussion of all the mechanisms of checking shown in Table 1. The facilities of Pascal and Ada are very widely known, those of the Intel 432 are discussed in [Intel, 1982]. We discuss the Protran example [Aird and Rice, 1983] some, as it is a newer and less widely known system.

2.2. Semantic Checking at Run Time

The PROTRAN system [Aird and Rice, 1983] is an extension of Fortran which adds numerous problem solving capabilities to Fortran. It uses as software parts about 100 of the programs from the IMSL library. These parts are invoked internally so that many of the syntactic matching problems are avoided; the system uses the syntax to select the appropriate software parts so that the matching is automatically correct. If the syntax is incorrectly specified, then the language processor catches the error and there is no attempt to use a software part.

PROTRAN does further checking at run time and formally identifies two types of run time errors: Problem Formulation and Numerical. A problem formulation error is where the problem set up is incorrect. For example, one might have specified to solve three differential equations, but the vector of initial conditions is of length 2. Or one might have SUM F(X); FOR(X = A,B,STEP) and STEP has the value 0 (or STEP = -.1 and A is larger than B). A numerical error is where the algorithm in the software part fails. Thus, if one asks to solve the linear system A*X=B for a matrix A and vectors X and B and if the matrix A is singular, then PROTRAN sets an error condition and marks X as undefined.

The PROTRAN system provides three options for actions when a problem formulation or numerical error occurs: IGNORE, WARN or ABORT. The first two are primarily for use in experimental codes; the default action is to abort the computation. If the IGNORE option is used and one has PRINT X where X is undefined, then one obtains blank output labeled as X.

3. AN INTERFACE SPECIFICATION STRUCTURE

In this section we present a detailed proposal for a general structure for interface specification. This structure allows for all the checking discussed above and provides a balance between flexibility, efficiency and completeness in interface checking. Several examples are given; realistic testing of this structure as not been made yet.

The encoding of the interface information has two conflicting objectives: it should be efficient and it should be extendable to arbitrarily complex data structures. The danger is, of course, that one uses 27 bytes to indicate that the following byte is a character. We propose a tree-like interface structure that:

(a) allows simple data items to be specified simply,

(b) explicitly exhibits the information that can be used for type checking at compile or load time,

(c) allows one to specify many complex data structures in terms of existing simpler ones,

(d) provides complete generality for data structures.

3.1. Definition of the Structure

The specification structure has, conceptually, three levels:

```
Level 1:  <Type> + <L1> + <Size>
Level 2:  <Length> + <L2>
          + <Data Structure> + <Qualifier>
Level 3:  <Data>
```

We assume that the computing environment uses bytes; other hardware can be accommodated by packing or by replacing bytes by words. The elements on each level are defined as follows:

<Type> = One of a small set of basic types encoded into one byte.

This is usually the type of the lowest level element in the data structure. We suggest that the types should be

```
CHARACTER        BIT         BLANK
INTEGER          REAL        MIXED
LIST
```

The MIXED type is for more complex data structures whose specifications are given at the second level. The BLANK type is similar in that the contents of the data are guaranteed to be that specified on the second level, even though the second level specification is not detailed. This type is intended for situations where efficiency requires that data not be moved or reformatted unnecessarily and it has already been checked. The LIST type is for a list of items, each of which is specified at a lower level.

<L1>
- 0 means the size of the data is given by <Size>
- 1 means that the size of the data is given given by the integer in the next <Size> bytes of the level 1 specifications

<Size>
- an integer whose functions is described described above

<Length>
- an integer, the number of bytes in the second level specification.

<L2>
- 0 means the data structure is in the <Data Structure> as a stan- dard data structure.

<Data Structure>
- an integer. If <L2>=0 this integer included one of the 128 standard data structures; otherwise this integer gives the number of the following bytes that give the name of the data structure (in characters).

It seems that 128 is more than enough standard data structures, but the list is perhaps longer than one might initially guess. For example, the standard data structures should include

```
2-D-ARRA        NAME-LIST   EXPRESSION-TEXT
3-D-ARRAY       MATRIX      DIRECTORY
UNASSIGNED      VECTOR      ARGUMENT-LIST
CODE-SEGMENT    FILE
PROCEDURE-OBJECT-CODE
```

which illustrates the need for many data structures beyond the usual ones.

<Qualifiers> - a set of information that is appropriate for each data structure.

Its length is not fixed, for example the qualifiers for a 1-D ARRAY should be

Name, lower-index-bound, lower-index-range, upper-index-range, upper-index-bound.

The qualifiers for MATRIX should be

Name, storage-format, property, #rows, #columns, row-range, column range.

And the qualifiers for NAME_LIST should be

Name, #pairs, name-lengths, value-lengths

In addition to the normal qualifying information, every data type also has two final "special" qualifiers. The next to last one is an "element" which allows replacement of the normal element specification by a new specification for the element using the two levels of this structure. The last one is an integer which allows addition of the indicated number of qualifiers to the data structure.

<Data> - The actual data. As illustrated below, this structure is recursive, so that the data may contain combinations of data structures specified by the two levels of the interface structure.

3.2. Six Examples

It is tedious to illustrate all the possible combinations of those definitions; we give six example specifications of data: (1) two real numbers, (2) a complex composite data structure, (3) a tree of real band, positive definite matrices, (4) a matrix with elements of a binary tree of character strings, (5) the argument list for a procedure, and (6) the list of arguments for the same procedure, plus a list of names associated with the tree. We use a verbose form of the specification to make these examples readable for this paper. In an actual system a compact notation would be adopted and a general notation for hierarchies of levels (recursion of specifications) used.

Example 1.

Two real numbers: 1.321 and 48.695

```
Level 1:    REAL + 0 + 2
Level 2:    0
Level 3:    .1321E+1, .48659E+2
```

Example 2.

A general region in a rectangular domain with a set of grid lines. This actual example consists of a set of two real procedures with two real arguments, one 2-dimensional integer array, three 1-dimensional real arrays, two 1-dimensional integer arrays and one 1-dimensional character array. This is an example of a special data structure created within this framework.

```
Level 1:    MIXED + 0 + 1
Level 2:    20 + 1 + 11 + "REGION-GRID",
            <I1>, <I2>,..., <I9>
Level 3:    <D1>, <D2>,..., <D9>
```

Here <I1>, <I2>, etc. are single byte codes for one of the 9 elements of the data structure. If <I1>, <I2> = X,C for the X-procedure and the character array, then <D1> is the data

```
REAL + 0 + 1
9 + 0 + PROCEDURE + "X", 2,REAL,
     REAL, OBJECT, 812, 0, 0
812 bytes of machine code
```

and <D2> is the data

```
CHARACTER + 0 + 1
7 + 0 + 1-D ARRAY + "TYPES", 0,
     0, 98, 250, 0, 0
98 character strings
```

Example 3.

A binary tree BMAT of depth 5 of real, positive definite, band matrices of order 40, range 20 and bandwidth 5.

```
REAL, 0, 1
4 + 0 + BINARY TREE + "BMAT", 5, 1, 0
REAL + 0 + 1
10 + 0 + MATRIX + BAND, POSITIVE
      DEFINITE, Ø, 5, 5, 40, 20, 0, 0
63 matrices (= 27,720 real numbers)
```

Example 4.

A band matrix BMB of order 40 and range 20 with band width 5 of (a) binary trees of character strings and (b) a list of up to 6 names of authorized users for the trees.

```
LIST + 0 + 1
9 + 0 + MATRIX + BAND, "BMB", 0, 5, 5,
     40, 20, 2, 0
CHARACTER + 0 + 1
5 + 0 + BINARY TREE + Ø, 5, 0, 0
CHARACTER + 0 + 6
5 + 0 + LIST + Ø, 6, 0, 0
440 pairs of 63 character strings
     plus 6 character strings
    (= 30,360 character strings)
```

Example 5.

The argument list for the linear equation solver M_SOLVE is of the form

```
MATRIX = <Name>,
SOLUTION = <Name>,
RIGHT_SIDE = <Name>,
ORDER = <integer_expression>,
ACCURACY = <key_word>
```

The specification of this argument list is:

```
LIST + 0 + 1
10 + 0 + ARGUMENT_LIST + "M_SOLVE",
         5, MATRIX, MATRIX, MATRIX,
         INTEGER, CHARACTER, 0, 0
```

Example 6.

The set of arguments for M_SOLVE could be gathered into the following list:

```
LIST + 0 + 5
5 + 0 + LIST + Ø, 5, 0, 0

REAL + 0 + 0
10 + 0 + MATRIX + "A", FULL, SYMMETRIC,
      100, 100, 20, 20, 0, 0
400 real numbers of the matrix "A"
REAL + 0 + 0
10 + 0 + MATRIX + "X", FULL, 0, 100,
      10, Ø, Ø, 1, 0
REAL + 0 + 0
1 + 0 + UNASSIGNED
(No data for the output argument) X
REAL + 0 + 0
10 + 0 + MATRIX + "B", FULL, 0, 100,
      10, 20, 2, 0, 0
40 real numbers of the matrix "B"
INTEGER + 0 + 1
0
20 (= the order of system solved)
CHARACTER + 0 + 1
0
HIGHACC (= keyword for high accuracy)
```

4. PROBLEM FORMULATION CHECKING

The underlying theme of interface checking is that the user cannot be trusted to use the software correctly. This concern is particularly high when large numbers of software parts are being invoked indirectly. Syntactic checking is the simplest and has the most pay-off. Experience shows that further checking is needed in order to provide really high reliability and thus we have been led to the elaborate specification structure presented in this paper and the related checking. Here we carry this theme forward to problem formulation checking. That is, we examine the entire set of inputs a software part to see if they define a well posed computation. We illustrate the situation with the linear equation procedure M_SOLVE introduced in Example 5 above, similar situations occur in many other computational areas.

Many software parts are somewhat generic in nature, for example, a part for sorting might sort integers, reals or character strings. A more complex part is one that solves linear systems of equations (such as M_SOLVE whose argument list is specified in the fifth example above). This part is generic in the sense that it

solves linear systems of different types (REAL, COMPLEX or DOUBLE PRECISION). Furthermore, the matrices involved must be compatible in size (the row-ranges of A and B must agree and the row and column bounds of X must be as large as the row and column ranges of B). The checking of types might or might not be possible at compile time, but the compatibility in size can only be checked at run time when M_SOLVE is invoked with actual arguments.

The size compatibility checking illustrated above is easily done by the prologue of M_SOLVE using the information given in the specification of Example 6 above. More subtle is the problem of checking the validity of the SYMMETRY specification. Given this specification, the procedure M_SOLVE should use an algorithm that takes advantage of the symmetry to reduce the computational work by half. If the standard algorithms are used directly, there is no checking and no "failure" if the matrix A is mismarked as SYMMETRIC. A robust software part would, in fact, perform this checking in addition at a reasonable computational cost. The work of solving this system is order $N^3/6$ for a N by N matrix while the checking requires work of the order $N^2/2$.

If the matrix A were specified to be POSITIVE DEFINITE, then an even more difficult checking problem arises. To check that A is actually positive definite is a computation equal to that of solving the linear system and it is unreasonable for the part's prologue to check this property. If the Cholesky algorithm is used, it will fail in a specific easily detectable way and no special check is needed. However, one might also use SOR iteration on the linear system as it always converges if A is positive definite. If A is not positive definite, then the iteration may continue indefinitely. This would, at least, eventually be identified as a failure and thus signal that the matrix A was not positive definite.

There are, of course, matrix properties which could be specified that would (a) allow very efficient solution of the system, (b) be very expensive to check, and (c) cause no obvious or easily computable failure condition to occur. An example of such a property is tensor product where ADI iteration is applicable for an extremely efficient solution method. Note that the naive test of substituting the computed solution into the linear system to see if the equations are satisfied is not a reliable checking procedure. However, more sophisticated versions of this approach (e.g., using sensitivity analysis) can provide high (but not complete) reliability.

We see, as one would expect, that it requires progressively more effort to provide higher and higher levels of reliability in the composition of software parts. Absolute reliability requires infinite effort in general, although it might be achievable in some small areas of computation. Those who strive for absolute reliability should be amused by the article [Davis, 1972] on the nature of mathematical proofs. Davis shows that even proving that the integer C is the sum of the given integers A and B is fraught with pitfalls.

5. TRADE-OFFS IN INTERFACE CHECKS

5.1. Gains: Higher Reliability and Faster Software Production

The primary goal of a software parts technology is faster and cheaper software production. The goal of an elaborate interface specification structure is to achieve high reliability also. We take as axioms that these goals are justified and believe that the interface structure proposed here provides high reliability. We make two additional points here: (a) There is an important situation when interface checking and its attendant costs are not required for high reliability, and (b) The economy in software production is for the users of software parts, not the creators.

Interface checking is needed because a part must be prepared for use everywhere and thus cannot "trust" that its input is correct. However, there is an important situation where the input can be "trusted". That is when several software parts are composed to form a larger part. The principle of modular software construction leads to the component parts retaining their identity within the larger part, so we would have the epilogue of one part putting the output data into a particular form and then the prologue of a second part methodically checking that this data is in the same particular form. Thus we visualize an "optimization" phase in software production using parts, one where a new part has been constructed and then redundant processing and checking at the interfaces is removed to improve efficiency. The structure of parts as prologue + nucleus + epilogue facilitates this optimization.

One should expect a software part to cost 5 to 10 times as much to create as a specific instance of it in a particular application. A software part must be designed, created and validated in complete generality; a specific instance exists in a narrow scope of application and thus need only be correct therein. A

software part must be documented so that users from widely different backgrounds can understand what it does and how to use it; a specific instance needs only be documented within its context of use. A software part must have its performance measured systematically and this information (along with much more) put into a catalog entry; a specific instance will probably not have its performance measured at all. Recall that a major source of inadequate software parts has been code lifted from a particular application and stamped as "general purpose" with only superficial changes.

5.2. Costs: Bulkier Data, Interface Overhead and Slower Execution.

There is no doubt that reliability will cost more; the question is: how much more? Interface overhead manifests itself as slower execution, so the extra costs are in the use of more memory and more CPU time. We believe the extra memory costs are modest and that the extra CPU time costs are not. We discuss tactics to minimize the costs of reliability, but note that these costs are inherently high. We believe that they are no higher in a software parts technology than elsewhere.

Data becomes bulkier because they are tagged with information to be used in checking. For a single number, this might double or triple the memory required. However, most bulky data are of some very systematic nature (e.g. a vector of 10,000 real numbers or a file of 10,000 identically structured records). The interface specification structure presented here allows one to tag these data "all at once" and at a small penalty in memory. It is true that programmers can be very inefficient here, the lazy one can determine how to tag one number and then use "recursion" to tag the 10,000 numbers is a vector instead of determining how to tag the vector itself. Overall, we do not expect this tagging to increase memory requirements significantly.

The overhead of interface checking can be significant. All one has to do is to put two or three small parts in the inner loop of some major computation. One can easily arrange to spend 50 to 90 percent of the CPU time in interface checking. This cost might be perfectly acceptable in prototype code development. It might be totally unacceptable in a production code and the interface overhead might be removed using the "optimization" process discussed above. The interface overhead that cannot be removed by optimization is probably checking that is essential to the reliability of the software and any approach would be costly. For example, in the inner loop example just mentioned, if previously "unseen", "external" data enters which must be checked, then one cannot optimize away the checking and retain reliability.

The following statement is widely doubted but supported by much observed evidence:

> "Expect to pay as much to check the correctness of an answer as to obtain the answer in the first place"

There are gross exceptions to this statement in both directions, but it underscores that run-time checking of problem formulation and solution correctness should be expected to be costly. Indeed, if one is not too concerned about correctness, one can achieve great efficiency and reliability by having all programs produce 1,709 as their result. An example of an important area where such costs are very high is in the solution of differential equations. These problems permeate programs for control (e.g. in robots, refineries, airplanes and nuclear power plants) and the best efforts so far have not achieved correctness checking in as little time as solving the problem. Here too, there is opportunity for optimization. One is usually dealing with the semantics instead of the syntax of the computation, so it is not as straightforward. Nevertheless, the knowledgable person is usually able to exploit the production use context to improve the efficiency of run-time solution checking.

5.3 Location: Compile-time, Load-time or Run-time?

As a general principle, one should do the checking as early as possible. Thus, the compiler should check as much as it can, but since it processes neither the whole program nor its data, there is only so much it can do. The loader has access to all the subprograms of a program, so it can complete the syntatic checking left undone by the compiler. Still, much checking must be done at run-time and this is the checking that is potentially the most expensive. One cannot check the validity of the input to a linear equation solver unless one has the input and attempts to solve the system.

We note in increasingly many computing environments that it is difficult to distinguish among run-time, compile-time and load-time. An interpretative language often translates the input into an internal code before executing the program; some of the checking could be done during the language translation instead of during

execution. Other systems have multiple levels of language processing; one can realistically visualize systems with 4 languages (e.g. ELLPACK to PROTRAN to Fortran 77 to C) where various semantic and syntatic checking is appropriate for each one. The user would think that he had simply run a short program and had the results displayed to him.

6. SUMMARY

A software parts technology should allow software to be produced more inexpensively than the current "handcrafted" or customized approach. The ingredients of this technology include a large supply of useful parts, a catalog of the parts, and mechanisms for composing or interconnecting parts, to form complex components. Of these, the interconnection mechanisms have received the least attention by researchers and are not well understood.

There are many issues which need to be addressed by a parts interfacing mechanism. This paper has addressed some of these. In particular, we have examined the levels at which these interfaces occur and we have proposed a mechanism for use in a parts oriented environment. This mechanism is based on data streams in which our data elements are labeled with a specification header. The trade-offs between efficiency and realibility have been discussed. What is required in an implementation, so that some of these interface issues can be evaluated.

We believe that a software parts technology will emerge, because of economic motivations. We also believe that the parts interface issue will be a major problem to be overcome as this approach to software development progresses.

REFERENCES

Aird, T.J. and Rice, J.R., PROTRAN: problem solving software, Advances in Engineering Software, 1983, to appear.

Batz, J., Cohen, P., Redwine, S. and Rice, J., Technical Aspects of the Application Specific Task Area, IEEE Computer, 1983, (Special issue on the STARS Project) to appear.

Comer, D., Rice, J. Schwetman, H., and Snyder, L. Project Quanta, CSD-TR 366, Computer Science Department, Purdue University, 1980.

P.J. Davis, Fidelity in mathematical discourse: Is one and one really two?, Amer. Math. Monthly, 79, 1972, 252-263.

Department of Defense, Software Technology for Adaptive, Reliable Systems (STARS), Program Strategy, Department of Defense, 1 April, 1983.

IMSL Library Reference Manual, IMSL, Inc., Houston, TX, 1979.

Intel, Intel 432 Reference Manual, Intel Corp, CA, 1982.

UNIX Time-Sharing System - special issue, The Bell System Technical Journal, 57(2), July-August, 1978.

Wasserman, A.I. and Gutz, S., "The future of programming", Communications of the ACM, 25(3), 1982, 196-206.

XM-80 v1.2 Language Reference Manual, Scientific Enterprises, Inc., Wilsonville, OR, 1981.

AUTHORS

John R. Rice received his Ph.D. from the California Institute of Technology in 1959. He joined Purdue University in 1964 as Professor of Mathematics and Computer Science after working in industry and government. He is Editor-in-Chief of the ACM Transactions on Mathematical Software and has current research interests in approximations theory, numerical software, partial differential equations and supercomputers.

Herb Schwetman received his Ph.D. in Computer Science from The University of Texas at Austin in 1970. Earlier, he received a B.S. from Baylor University in 1961 and an Sc.M. from Brown University in 1965, both in Mathematics. He has worked at IBM, The University of Texas Computing Center, and Boole and Babbage, Inc.

He has been at Purdue University since 1972. Currently, he is Associate Professor in the Department of Computer Sciences. In 1979, he was a Fullbright/Hayes Lecturer at the University of Helinski, Finland. He is a member of ACM and the IEEE/Computer Society. His current research interests include software engineering, models of computer systems, and performance evaluation.

The author mailing address is:

Department of Computer Science
Purdue University
West Lafayette, IN 47907