1989

Composition of Libraries, Software Parts and Problem Solving Environments

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

Report Number:
89-852

COMPOSITION OF LIBRARIES, SOFTWARE PARTS AND PROBLEM SOLVING ENVIRONMENTS

John R. Rice

CSD TR-852
January 1989

COMPOSITION OF LIBRARIES, SOFTWARE PARTS AND PROBLEM SOLVING ENVIRONMENTS

J. R. Rice*

Computer Sciences Department
Purdue University
CSD-TR-852
January 1989

Abstract

We consider the problem of creating very large software systems by composing existing software. As components of this process we consider software library elements, software parts and problem solving systems. These items are briefly defined and their principal characteristics given. We discuss the relationship of these items to objects in object oriented programming. The mechanics of the composition are discussed, the key issues are identified and the trade offs discussed. The principal tradeoff is, of course, programming effort versus execution time efficiency. We consider several examples and conclude that building very large software systems as a set of cooperating, somewhat autonomous, parts is a promising direction for research.

* This work is supported in part by the Air Force Office of Scientific Research under grant 88-0243, and the Strategic Defense Initiative under Army Research Office contract DAAL03-86-K-01606.

COMPOSITION OF LIBRARIES, SOFTWARE PARTS AND PROBLEM SOLVING ENVIRONMENTS

John R. Rice*
Computer Science Department
Purdue University
West Lafayette, Indiana 47907, U.S.A.

I. INTRODUCTION

We consider the problem of creating very large software systems by composing existing software. As components of this process we consider software library elements, software parts and problem solving systems. These items are briefly defined and their principal characteristics given. We discuss the relationship of these items to objects in object oriented programming. The mechanics of the composition are discussed, the key issues are identified and the trade offs discussed. The principal tradeoff is, of course, programming effort versus execution time efficiency. We consider several examples and conclude that building very large software systems as a set of cooperating, somewhat autonomous, parts is a promising direction for research.

II. SOFTWARE LIBRARIES

A software library is a collection procedures which are organized loosely to cover an application area. In the ideal case, this is a well structured collection of independent units which can be included into a particular programming language code to implement all the important operations of an application area. There is little or no *context* implicit in the library beyond that of the programming language with which the library is used. The input, output and operation of each library procedure is described in separate documentation. The *conceptual level* of a library element is strongly influenced by the level of the programming language. For example, in Fortran 77 one can pass functions to a

* This is work is supported in part by the Air Force Office of Scientific Research under grant 88-0243, and the Strategic Defense Initiative under Army Research Office contract DAAL03-86-K-01606.

procedure only by adhering to conventions specific to each procedure, one must supply "redundant" auxiliary information about arrays, and allowable data structures are severly limited. A library is *static* in the sense that the user is presented with library and has no way to modify or add to the procedures in it.

The key issues in library design and use are:

Coverage:	How complete is the set of procedures provided? This determines the power of the library.
Retrieval:	How does one find the appropriate procedure? Individual libraries with 500-1000 procedures are becoming common. A user may have easy access to many thousand library procedures and find that dozens – or even hundreds – potentially implement the operations he needs.
Documentation:	How does one understand the procedures and their use? Preparing easily understood documentation is very difficult.
Interfaces:	How does one create an interface to a library procedure? The user must write code in the target programming language which creates the input and output data structures and which invokes the procedures. This code is usually lengthy compared to invocation of the library procedures.

The first three issues are under the control of the library builder. In the past 15 years we have seen these issues addressed strongly by IMSL and NAG resulting in much improved libraries being created. The interface is still the responsibility of the user and is inherent in the library concept. We use the term *procedure team* to denote a library that has been well structured with good coverage, retrieval and documentation.

III. SOFTWARE PARTS

Software parts are defined within a standard framework of a specific application area, e.g., linear algebra, statistics, compressor design or compilers. This standard framework provides a uniform environment for the data structures, procedures and terminology used in the application area. Most technologies (e.g., plumbing construction, electronics, mathematics) have created very rich frameworks, so much so that one has to learn a jargon in order to communicate. That is the point, once the jargon is mastered then one can communicate quickly with everyone about most things of importance. Software engineering has not yet created such frameworks of note, but we can

expect them as the field matures. The operation of a software parts technology is illustrated in Figure 1. A software parts technology greatly facilitates the warehousing, cataloging, retrieval and composition of software components. See [Batz et. al., 1983] and [Rice and Schwerman, 1983] for further discussion.

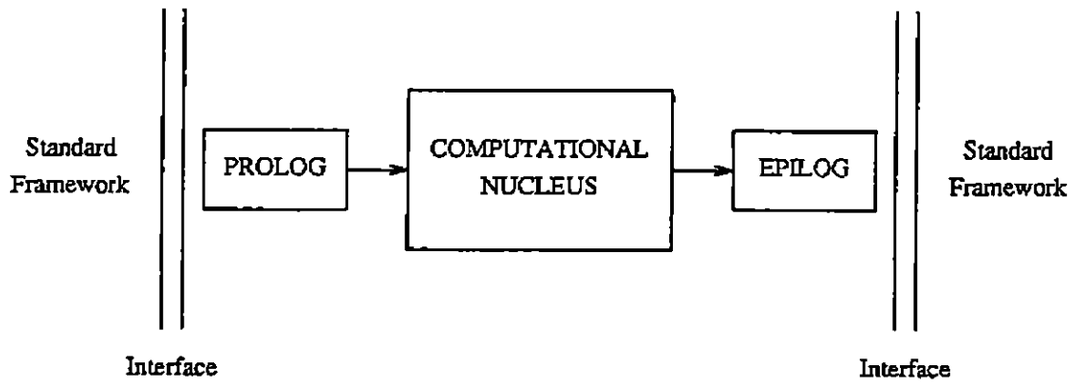

Figure 1. Simple model of a reusable software part. The prolog and epilog are responsible for conversions between the standard framework and the computational nucleus.

A software parts technology is the natural evolution from a library. The standard framework provides the conceptual framework including a variety of standard data types which all software parts can use and which are not constrained by a programming language. As a library, it is still static (at least for the user) but it can be multi-level. That is, there may be parts that operate at a very low level with much detailed specification required while there may be others that are at a very high level. Thus one part could define a piece of an engine valve using detailed measurements while another could solve for the temperature on the whole engine block with one simple command.

The key issues for software parts include those for a library: coverage, search, documentation and interfaces. It is more crucial that the coverage be good. The standard framework makes search and documentation much simpler. An additional key issue is

Composition: How does one put the parts together?
A software parts technology can use an existing programming language as a parts composition systems, but this is likely to be awkward and inadequate. Some specialized facilities to assist parts composition is almost surely required for an effective software parts technology.

Note that a software parts technology cannot be created independently by a single person or organization. It relies on the application community agreeing on the standard framework.

IV. PROBLEM SOLVING ENVIRONMENTS

A problem solving environment (PSE) is the natural extension of a software parts technology. It has the standard framework, all the application data structures (variables), declarations (problem definition statements), and operators (problem solving statements) all wrapped up in a single, coherent system with its own language and interface to the user. See [Ford and Chatelin, 1987] for more information and examples. A PSE is holistic in nature, it provides everything necessary within its application area. Thus it is (normally) at a very high level with explicit context.

The previous, somewhat separate, key issues of coverage, research, documentation, interfaces and composition are now combined into single one: how good is the PSE? All these issues must be addressed well or the PSE is flawed. There is another key issue

World view: Does the PSE assume it is the only software?
Does it assume it interacts only with a human?
Many PSEs do not provide free access to external software systems. Some are not easily suspended while another is running (when one exits, the system dies and loses all its status information).

The world view is not so important for libraries and software parts because access to the external environment is usually provided by the programming language or parts composition system.

Figure 2 shows how these three software items are related.

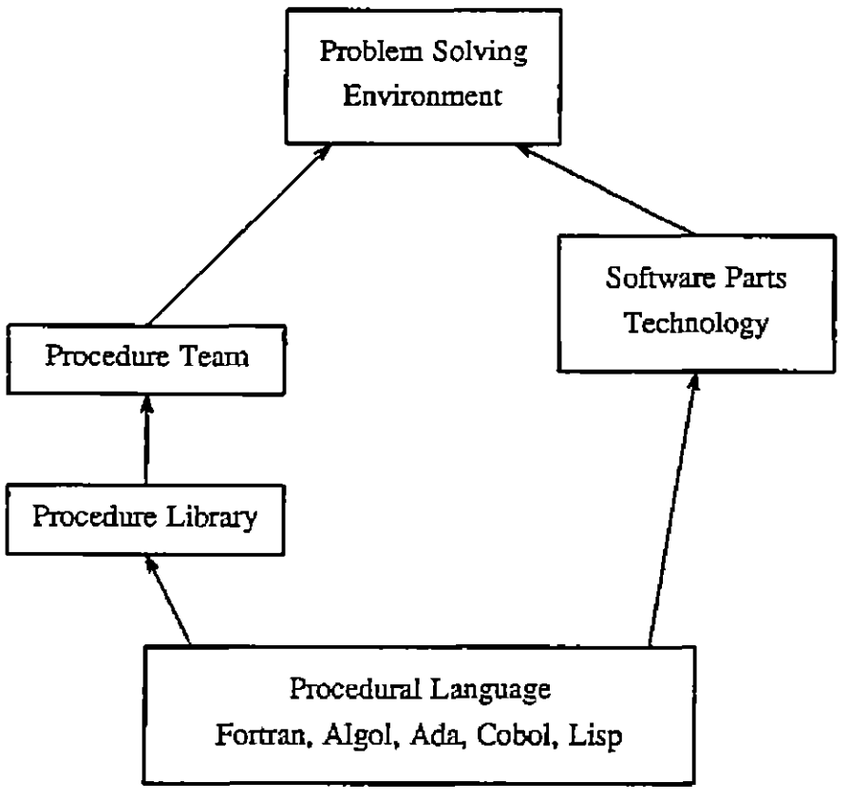

Figure 2. The hierarchy of software components in the construction of large software systems.

V. OBJECTS

Recently *object oriented programming* has been a popular vehicle for software development. The key idea is to make objects completely self-contained (as are software parts) and more general than procedures. When an object receives a message (is invoked with arguments) then certain actions result. An object differs from a procedure in three important ways. First, its actions are not limited to producing output values, one might have output values or side effects (e.g., it starts a new page on the printer with the current date and page number at the top). Second, it does not automatically assume the thread of control in an exclusion manner. One object invoking another might continue to take actions and the invoked object need not "return control" to the invoking object. Finally, an object may respond to many different messages, it does not have a unique "calling list of arguments" or even a single "entry point". Note that objects are not static, it is possible for one object to create another immediately. See [Cox, 1986] for further discussion.

Thus objects are generalizations of library procedures and software parts whose principal properties are essentially orthogonal to those of objects. One can have *object libraries*, *object teams* and a software parts technology based on objects. One can modify Figure 2 by replacing the word "procedure" by "object" and it is still valid. One can mix object oriented programming with conventional library mechanisms because a procedure is a particularly simple object.

VI. EXAMPLES

We discuss some simple examples of software systems.

1. *Directory system for UNIX*

Is it a library?	No. There is no general processing capability.
Software parts?	No.
PSE?	Yes. It has a standard framework of a tree and types (directory, files, executable, ...). There are very simple problem solving statements: *move, remove, copy, list, change access rights.*
World view?	None.
Object oriented?	Yes. The directory commands regard the contents as objects.

Note that within the larger system, one can use editors to create or modify the directory system variables in arbitrary ways. This is not possible within the directory system itself.

2. *MACSYMA*

Is it a library?	No.
Software parts?	No. But MACSYMA could be implemented using software parts.
PSE?	Yes.
World view?	Assumes it is interacting with a human at a terminal.
Object oriented?	No.

3. *ELLPACK*

Is it a library?	No. But there is one inside it.
Software parts?	No. But it is implemented this way, with its own narrow standard context.
PSE?	Yes.
World view?	Standard ELLPACK is batch oriented but not directly invokable by other software. It may "exit" and "return" using access to Fortran. Interactive ELLPACK assumes it is interacting with a terminal.
Object oriented?	No.

4. *NASTRAN, SPSS and the IMSL Library*

Is it a library?	Yes.
Is it a team?	Yes.
Software parts?	No. The standard framework is lacking.
PSE?	No. But some PSEs have been built to interface these libraries (*Adam* for NASTRAN, conversational SPSS, and *PROTRAN* for IMSL)
World view?	None. Free access to procedures via Fortran.
Object oriented?	No.

VII. TECHNICAL ISSUES IN COMPOSING SOFTWARE

Our goal is to study composing existing software elements in order to create very large software systems. There are four general questions to consider:

1. *How much effort is required?*
2. *What technical barriers are incountered?*
3. *What are the savings in software production costs?*
4. *What are the losses in execution efficiency?*

Examples of barriers to software composition are listed.

A. *Machine specificity of software.* Much software is dependent on specific machine features. Thus to compose one component written in Lisp running on a Symbolics and another written in a vector Fortran running on a Cyber 205 requires either:

(i) Rewriting software, e.g., porting both components to a machine that runs Lisp and Fortran well.

(ii) Creating a distributed composition system involving multiple machines.

B. *Language specificity of software.* Libraries are language specific, the IMSL library is for Fortran use. Software parts are likely to be programming language specific although they could be commands in an operating system. To compose these procedures requires either the translation of whole libraries (which may be quite feasible) or the development of a set of language interface facilities (which also may be quite feasible).

C. *Single user design.* Most problem solving environments are designed as the "entire computing environment". Examples of this are MACSYMA, MATLAB, Interactive ELLPACK, and the Symbolics system. Substantial modifications may be necessary to make these PSEs into one of a collection of software components cooperating in a larger system. Closely related to this barrier is that some systems assume that they interact with a human. This assumption can be difficult to remove, e.g., computations may depend on the slow response time of people or a system might depend on a human's ability to recognize features in a graph or display.

D. *Low level incompatibilities.* Examples here are assumptions about word lengths, memory sizes, character representations (ASCII or EBCDIC), array storage formats or the number of open files possible. While these incompatibilities are mundane, they can be major barriers to software composition.

We are actually more ambitious than just wanting to compose software, we want to create systems that cooperate in solving complex problems. In many applications one sees that combining a set of software systems results in solving the applications, i.e., there is a direct sequence of steps from the problem to the solution. But suppose we have a collection of software components that each solve one part of a complex problem, can they be combined to solve the whole problem? The answer, of course, depends on the particular problem, the particular collection of components and the cooperation strategy used. In the next sections we present three examples of such applications, but the additional technical issue raised is:

5. *What framework is suitable to allowing software components to collaborate in an application?*

VIII. COLLABORATORY SOLUTION OF AN ODE PROBLEM

We present an artificial problem to illustrate the idea of collaborating software components for solving a non-standard problem. The problem is compute two functions $u(x)$ and $v(x)$ which satisfy:

Differential Equations:

$$\begin{aligned}
u'' &= (5+x^2)\,u + \sin(xu+v) - u'v(1+x^2) && x\in [.2,.6]\\
u'' &= (4+xv)\,u + e^{xy}/(1+xv) - .2u'(1+v^2) && x\in [.6,1.0]\\
v'' &= (3x^2+u)v && x\in [.2,.75]\\
v'' &= (10+u/x)v - .4(\sin(x+v)+3.5u)v' && x\in [.75,1.0]
\end{aligned}$$

Interface Conditions:

$u(x)$ continuous at .6	$u'(.75) = 1$
$u'(x)$ continuous at .6	$v(.6) = 1$
$v(.2) = u(.2)$	$u(1) = 2$
$v(x)$ continuous at .75	$v'(1) = 0$

There are 12 equations here, 4 differential and 8 algebraic. It is unlikely that we find an existing software item that solves this problem directly. Indeed, it is not clear that this problem is well posed in general.

The tools we would use to solve this problem are ODE solvers, say, IVPRK and BVPFD from the IMSL library. Suppose we take these and create four ODE solving objects:

Object D1:	Input.	$u(.2)$ and $u(.6)$ and $v(x)$ on [.2,.6]
	or	$u(.2)$ and $u'(.2)$ and $v(x)$ on [.2,.6]
	or	$u(.6)$ and $u'(.6)$ and $v(x)$ on [.2,.6]
	Output.	$u(.2)$, $u(.6)$, $u'(.2)$,$u'(.6)$,$u(x)$ on [.2,.6], residual norm on [.2,.6]
Object D2:	Input.	$u(.6)$ and $u(1)$ and $v(x)$ on [.6,1.0]
	or	$u(.6)$ and $u'(.6)$ and $v(x)$ on [.6,1.0]
	or	$u(1)$ and $u'(1)$ and $v(x)$ on [.6,1.0]
	Output.	$u(.6)$, $u(1)$, $u'(.6)$, $u'(1)$, $u(x)$ on [.6,1], $u'(.75)$, residual norm on [.6,1.0]
Object D3:	Input.	$v(.2)$ and $v(.75)$ and $u(x)$ on [.2,.75]
	or	$v(.2)$ and $v'(.2)$ and $u(x)$ on [.2,.75]

	or	$v(.75)$ and $v'(.75)$ and $u(x)$ on [.2,.75]
	Output.	$v(.2)$, $v(.75)$, $v(.6)$, $v(x)$ on [.2,.75], residual norm on [.2,.75]
Object D4:	Input.	$v(.75)$ and $v(1)$ and $u(x)$ on [.75,1.0]
	or	$v(.75)$ and $v'(.75)$ and $u(x)$ on [.75,1.0]
	or	$v(1)$ and $v'(1)$ and $u(x)$ on [.75,1.0]
	Output.	$v(.75)$, $v(1.0)$ and $v(x)$ on [.75,1.0], residual norm on [.75,1.0]

We also create simple objects that "solve" the eight interface conditions individually. Two of these we impose everywhere, once and for all, namely, $u(1) = 2$, and $v'(1) = 0.0$. The corresponding two objects simply reset these values whenever they are invoked. There are four simple objects specified in the following table.

	Associated Condition	Input	Output = Action
I1:	u continuous at .6	$u(.6)$ from D1 and D2	$u(.6)$ = average of inputs
I2:	u' continuous at .6	$u'(.6)$ from D1 and D2	$u'(.6)$ = average of inputs
I3:	$v(.2) = u(.2)$	$u(.2)$ from D1 $v(.2)$ from D3	$v(.2) = u(.2)$ = average of inputs
I4:	v continuous at .75	$v(.75)$ from D3 and D4	$v(.75)$ = average of inputs

It is not obvious how to "solve" the interface conditions $u'(.75) = 1$ and $v'(1) = 0$. We create two objects which, intuitively, improve the solutions to these conditions.

Object I5.	Input.	$u'(.75)$, $v(x)$ on [.6,1.0]
	Output.	$u(.6)$, $u(x)$ on [.6,1.0], residual norm on [.6,1.0]

This object solves the ODE on [.6,1.0] to satisfy the conditions $u'(.75) =$ and $u(1) = 2$.

Object I6.	Input.	$v(.2)$, $v(.75)$, $v(.6)$, $u(x)$ on [.2,.75]
	Output.	$v(.2)$, $v(.75)$, $v(x)$ on [.2,.75], residual norm on [.2,.75]

This object modifies the values $v(.2)$ and $v(.75)$ by half the change detected by the solutions (i) with $v(.75)$ as given and $v(.6) = 1$ for the new $v(.2)$, and (ii) with $v(.2)$ as given and $v(.6) = 1$ for the new $v(.75)$.

The cooperation mechanism that we propose is as follows. First, make initial guesses for $u(x)$, $v(x)$ and the quantities in the interface conditions and compute the residuals for all 12 equations. Then sequentially apply the objects whose residuals are the largest until the residuals become small enough. In a parallel computing environment, one can apply several objects concurrently. It is clear that one should explore more sophisticated strategies in selecting which objects to invoke when. However, in the background is the belief that many physical situations are naturally stable so that good mathematical models of them should be stable also and many reasonable cooperating strategies are effective.

IX. ADDITIONAL EXAMPLES

We briefly describe three applications where composition is or should be quite effective.

1. *COMPUTING ABOUT PHYSICAL OBJECTS*

This project is to explore how to construct computer systems that accurately model a broad range of behaviors that we observe in the physical world. Shapes change. Some are smooth and beautiful, others are angular and functional. Unanticipated interactions take place, billiard balls bounce off one another, a robot arm jerks and breaks a lever, electric current boils water and a whistle blows. In exploring how to focus the computer power needed to create such a system, we realize that a huge heterogeneous software system must be created. Successes have already been achieved by composing large, unrelated software systems running on different machines. Effective techniques have been devised to access the IMSL library from languages other than Fortran. For further discussion see [Bajaj et. al., 1988].

2. *COOPERATING REPLICATED ELLPACKS*

We have already developed Parallel ELLPACK where a domain is subdivided into many parts and each is given to a single node of our 128 processor NCUBE machine. Then the discretization is made and the resulting linear system solved using the parallelism of the machine. See [Houstis and Rice, 1989] for further discussion. This approach is quite effective for dividing a single partial differential equation problem into many pieces to exploit parallelism. And the software system at each node is really a specialized, but essentially complete, version of the ELLPACK problem solving environment. However, this composition of PSEs is very uniform in nature. More complex physical applications require something different.

We visualize a network of cooperating replicas of the ELLPACK system. Each one is able to solve a single PDE problem on a single, not too complicated domain. The network is connected to reflect the underlying physical geometry which has been decomposed into fairly simple shapes. On each piece we have a single equation. There may be multiple equations for a single geometric piece, in which case we replicate the piece so as to retain one equation per piece. Then there are a large number of interface objects similar to those defined in Section VIII. There is a high level control of the cooperation between these software objects which direct the computation toward the solution of the whole problem.

3. *COMPOSING ENGINEERING SYSTEMS FOR DESIGN OPTIMIZATION*

In [Tong, 1989] a project is described which involves large engineering Fortran software codes for the design of jet engine components. Each code does a design analysis for a small part of the engine. These codes are specified by law so one cannot change even one line while composing them together. Tong's goal is to use systematic and heuristic optimization techniques involving a number of codes (the designs interact but the codes do not take this into account). He created a set of objects (in the technical sense), each with one of these codes as its computational nucleus. Tong then constructs a system using these objects which employs various "cooperating" optimization strategies in order to produce better designs. He reports good success, designs have been made which substantially improve on the best previous "human engineered" ones. Further, the elapsed time to make the designs is greatly reduced.

X. CONCLUSIONS

We have reviewed the hierarchy of software elements and discussed the potential to create very large software systems by composing large numbers of software elements. We conclude that this process can be very effective in many instances and we conjecture that the gain in software productivity and power is more than offset by the reduced execution time efficiency of systems built this way. We have introduced the concept of a network of cooperating software systems and conjecture that it can be a very effective approach to solving important scientific applications.

REFERENCES

C. Bajaj, W. Dyksen, C. Hoffmann, E. Houstis, J. Korb and J. Rice [1988] Computing about physical objects, *Proc. 12th World Congress on Scientific Computing*, IMACS, **4**, pp. 642-644.

J. Batz, P. Cohen, S. Redwine and J. Rice [1983], The application specific area, *IEEE Computer*, **16**, pp. 78-85.

B. Cox [1986], *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley.

B. Ford and F. Chatelin [1987], *Problem Solving Environments for Scientific Computing*, North-Holland.

E.N. Houstis and J.R. Rice [1989], Parallel ELLPACK, *Math. Comp. Simulation*, **31**, to appear. Reprinted in *Fourth Generation Mathematical Software Systems* (Houstis, Rice, Vichnevetsky, eds.) North-Holland, to appear.

J.R. Rice and H.D. Schwetman [1983], Interface issues in a software parts technology, in *Reusability in Software* (E. Biggerstaff, ed.), ITT Technology, pp. 129-137. Reprinted in *Software Reusability* (P. Freeman, ed.), IEEE Tutorial, Computer Society Press (1987), pp. 96-104.

S.S. Tong [1989], Coupling artificial intelligence and numerical computations for engineering design, *Math. Comp. Simulation*, **31**, to appear. Reprinted in *Fourth Generation Mathematical Software Systems* (Houstis, Rice, Vichnevetsky, eds.) North-Holland, to appear.