A Common Distributed Language Approach to Software Integration

Final Report

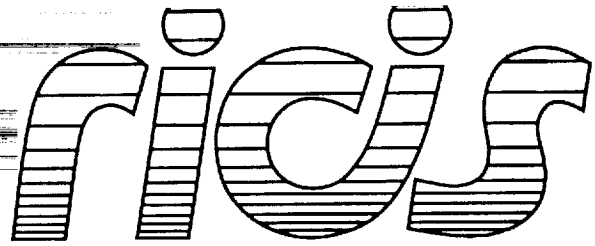
Charles J. Antonelli
Richard A. Volz
Trevor N. Mudge

The University of Michigan
and
Texas A&M University

August 9, 1989

Cooperative Agreement NCC 9-16
Research Activity No. AI.03

NASA Johnson Space Center
Mission Support Directorate
Mission Planning and Analysis Division



Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

A Common Distributed Language Approach to Software Integration

Final Report

Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Charles J. Antonelli, Richard A. Volz, and Trevor N. Mudge. Dr. Terry Feagin served as RICIS research coordinator.

Funding has been provided by the Mission Support Directorate, Mission Planning and Analysis Division, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Dr. Timothy F. Cleghorn, of NASA/JSC.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.

A Common Distributed Language Approach to Software Integration Final Report*

Charles J. Antonelli Richard A. Volz
Trevor N. Mudge

Center for Research on Integrated Manufacturing
Robot Systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109

Department of Computer Science
238 Zachry Engineering
Texas A&M University
College Station, Texas 77843-3112

August 9, 1989

Abstract

An important objective in software integration is the development of techniques to allow programs written in different languages to function together. We discuss several approaches to achieving this objective and present the Common Distributed Language Approachas the approach of choice.

*This work was sponsored by NASA JSC Grant No. NCC-9-16.

1 Introduction

One important objective in software integration is the development of techniques which allow programs to function together. By "function together", we mean the mutually cooperative execution of a number of programs to achieve a common goal.

We will need to define precisely what we mean by "programs" because we shall consider two different types of software components as our basic building blocks in this paper. When we say *program* we will mean a software component that stands alone, and is loosely coupled with other programs through files, pipes, sockets, and other types of input/output devices. When we say *subprogram* we will mean a software component which may or may not stand alone, but is tightly coupled with other subprograms through local and remote procedure calls. Subprograms are thus constituents of programs; to simplify the nomenclature, we shall always use the term *program object* to denote programs and subprograms collectively. We have identified the following attributes as distinguishing programs from subprograms:

- **Interaction Times.** Programs interact with each other more slowly than subprograms do. This is because independent programs must rely on traditional, relatively slow I/O communications interfaces. Even the fastest communications primitives cannot compete with the speed and efficiency of a local procedure call. Subprograms, because they can employ local procedure calls, can interact much more quickly.

- **Compatibility.** Programs and subprograms resolve data formatting incompatibilities differently. Programs write data into files and onto other output devices, while other programs read the data back and perform some type of conversion. This allows a much greater flexibility in data formatting. In contrast, subprograms must interact via procedure calls, so they not only must arrange to convert the data into a format that is mutually compatible, but the formatting of these data is usually limited to a small number of subprogram-defined data types.

- **Coupling.** Individual program execution is usually much more of a decoupled activity than subprogram execution. Programs which produce output can be run independently of other programs which use their data as input. Many runs of a producer can be performed, and several data

files generated, before the consumer programs need be run. Failures in consumers can be addressed by simply rewriting them and re-executing them on the same produced data. Subprograms, on the other hand, are much more tightly coupled, in that data shipped across a procedure call is much more short-lived, and failures in one or the other component usually force all of the subprograms to terminate and be re-executed.

- **Concurrency.** Programs can usually operate concurrently with other programs, spooling intermediate data in buffer areas. Subprograms, on the other hand, usually run synchronously, with a caller waiting until a called subprogram returns. Even remote procedure calls, in which the remote procedure is located on a different processor, usually enforce this synchrony.

- **Data Sharing.** If the programming language in which a set of subprograms is written supports the concept of globally visible data objects, then subprograms can share such global data in a straightforward manner. In addition, since the subprograms of a program are sequentially executed, problems caused by indeterministic concurrent access to data do not occur. On the other hand, programs usually cannot share data this easily; the underlying operating system and/or hardware architecture must be provided with mechanisms to allow one program to access data stored in another program. Concurrent access to such shared objects by different programs must then be controlled, for example via a monitor.

Thus there are advantages and disadvantages to both types of program objects; this means we cannot eliminate either category, and instead will analyze both.

We wish to place as few restrictions on the composition of these program objects as possible. In particular, this means we will not insist on the simplification that these objects be designed or written together. On the contrary, we feel the greatest benefit can be obtained if our methodology may be applied to program objects that were never intended to function together, that were designed to run in stand-alone mode on different machines with different operating systems. The bulk of currently existing software consists of specialized program objects tailored for a particular application. If we can develop a methodology that makes them usable as "building blocks" we can realize benefits in rapid prototyping, reusable software, and software integration without the need for

the continual tailoring and re-tailoring of supposedly general-purpose software packages to individual applications.

The rest of this paper is organized as follows. Section 2 examines the important issues and categorizes several possible approaches that may be taken in order to achieve our objective of getting program objects to work together. One of these approaches, which we call the Common Distributed Language Environment, is then selected for more detailed study in Section 3. Finally, we propose a candidate for this common language (Distributed Ada) and develop several methods for interconnecting programs using this language in Section 3.

2 Software Integration Issues and Approaches

Let us denote a program object written in some unspecified *foreign language* FL as a *foreign program* FP. Then, given a set of foreign programs $FP_1, \ldots, FP_n$, what must be done in order to connect them together?

This question is examined by first discussing several of the issues involved in making such connections among program objects. We do so to demonstrate the breadth and scope of the overall problem, and to separate areas in which automated procedures can realistically be expected to assist in this software integration process from those areas in which manual processing is required. Some of these issues, not amenable to automation, are not examined further.

Second, several approaches to solving the remaining issues through the interconnection of FPs are discussed. Here the goal is to determine methods of solving the problems associated with the previously identified issues. Again, two of the approaches are discarded because they require too much manual crafting.

Finally, we select the Common Distributed Language Approach for further study in future sections.

2.1 Issues

We have identified the following issues which must be dealt with in order to integrate arbitrary subprograms.

2.1.1 Syntactic Compatibility

An FP receiving data from another must be cognizant of the data formats used by that FP. For example, a receiver must be able to interpret correctly the value of an integer variable transmitted by a sender. The nontriviality of this problem in the light of varying word lengths, byte orderings, and more complicated data structures is obvious.

2.1.2 Semantic Compatibility

After solving the problem of syntactic compatibility, a receiving FP must correctly interpret the semantics of the arriving data. Elaborating on the previous example, consider the case in which the sending FP is a program which outputs strings whose meaning depends on the current state of the sender as well as the contents of the strings[1].

For example, consider an editor FP_1 whose inputs and outputs are connected to a driver FP_2. Suppose the driver FP_2 is presented with a string from the editor which consists of the characters "input error". If the editor is in a state in which it is accepting and reacting to commands issued by the driver, the meaning is clear: the previous command sent by the driver was not understood. However, if the editor is in a state where it is echoing portions of the edited text, then the meaning is entirely different: the string "input error" is part of the text being edited which is being listed on the editor's output, and there is no error associated with the transmission of this string.

Thus, the compatibility issue is not just a matter of assigning the proper mappings between data representations. After the receiver solves the problem of correctly interpreting the strings themselves, it must further assign them meanings utilizing information not present in the strings. Thus, if we consider the values being exchanged by the programs to be trees, then we are concerned at this level of compatibility with providing methods of understanding the forest.

2.1.3 Implementation Languages

Here we are concerned with the types of languages used in writing each of the FPs. There will obviously be several such languages, and possibly several dialects of each.

[1] In fact, this paradigm describes most user – operating system interfaces in use today.

The basic issue here is one of compatibility. Different languages have very different concepts of programs, data, data representations, control flow, and synchronization. Even identical languages are often secretly influenced by the type of machine and operating system supporting the compiler(s) and run-time system(s). Subprograms must match parameter types, ordering, and representations at the call interfaces. Programs must match byte ordering, data structure representations, and the correct matching of input with output operators.

2.1.4 Implementation Language Compilers

Along with each implementation language there will be a compiler or set of compilers required to translate this language into executable code. Each such compiler will likely accept a slightly different dialect of FP, provide a unique set of interfaces to the processor and/or operating system on which the FP runs[2], produce different intermediate and internal data representations, and produce code with different size, speed, and efficiency characteristics.

2.1.5 Configurational Variances

Given compilers which produce identical code for the same foreign language, differing implicit or explicit configurational requirements limit interoperability. As a classical example, consider the attempted migration of a FORTRAN program from one hardware configuration of a particular computer system to another in which different I/O devices are provided. Even though identical programs, compilers, libraries, and operating systems are provided for both systems, the program as written for the first configuration simply will not run on the second without modifications.

2.1.6 Architectural Variances

Here we take the variabilities mentioned in the previous paragraph and extend their scope to the architecture and thus the overall structure of the system intended to execute the FPs itself. The issue here involves removing ourselves from the implicit architectures that existed at the time the majority of the FPs were written – centralized processors – and consider the difficulties involved

[2]Assuming that the compiled code has the responsibility for managing resources. If it does not, the run-time system supplied with the compiler surely does, and the problem remains.

when distributed systems and massively parallel processors of all flavors must be considered as part of the target environment. For example, an Ada program written to execute on a traditional virtual memory platform (such as a Sun processor) will not be transportable to a non-virtual massively parallel architecture such as the NCUBE hypercube without extensive modifications, because of the great architectural differences that exist between these two systems.

2.1.7 Operating System Variances

Even when two machine architectures are identical, it is quite possible to develop quite different operating systems which exacerbate the difficulties of connecting together program objects not written for the same operating system. In particular, the system call interface of each operating system is likely to be very different. As a example, consider the massive differences between the UNIX[3] and VMS[4] operating systems and the resulting difficulty of connecting together two programs, one from each OS. As a less obvious example, consider the construction of a program out of subprograms drawn from the Berkeley UNIX and the System V UNIX environments. Now the differences are largely hidden, so much so that the job of constructing such a program seems deceptively simple at first glance, but reveals some deep architectural differences hidden beneath the surface. For example, Berkeley UNIX programs which utilize the `ftruncate()` function are not easily transported to System V UNIX implementations in which this library function does not exist[5].

2.1.8 Discussion

Since the first two issues (those having to do with the syntactic and semantic compatibility problems mentioned in the previous section) are inherently manual in nature, we do not consider them further in this paper. This is because it is inherently a creative process to analyze two FPs and determine their syntactic and semantic compatibilities, and this process thus must be manually rather than automatically performed.

[3]UNIX is a trademark of AT&T.

[4]VMS is a trademark of Digital Equipment Corp.

[5]It is not a simple matter to simulate this function under System V, because this operating system does not supply a file truncation primitive.

It is difficult to solve either of these problems automatically. Syntactic compatibility must be established by a pairwise examination of all program objects that are to be connected together, with the creation of the appropriate conversion subprograms. Several approaches are possible; in one such approach, the receiver of data is made responsible for their correct interpretation by performing the appropriate conversion. This type of conversion is most easily performed if a set of conventions is followed when such interfaces are designed, because then an automated tool can be employed to generate the appropriate conversions; it is much more difficult to generate these conversion subprograms for existing foreign program pairs, that is, pairs which were not originally designed to function together.

Semantic compatibility, which also must be established for all connected pairs of program objects, presents a different type of problem, since it is the states of the two connected program objects that must be synchronized. Since, in general, the state of each program object is independent of the state of the other program object in a given pair, is is exceedingly difficult to establish semantic compatibility automatically for existing foreign program pairs. Continuing the previous editor example, a program for achieving this semantic compatibility would first have to determine the number of states the editor could assume. It would then have to discover how the meaning of the output string "input error" changes with respect to the editor state. Finally, it would need to interpret occurrences of this string for the other program connected to the editor based on the editor state, and it would have to do this for all possible strings output by the editor. Different program pairs would naturally exhibit different internal states and resulting interpretations. It is clear that such semantic compatibility cannot, in general, be achieved automatically.

Accordingly, we concern ourselves chiefly with the language, configurational, architectural, and operating system variances in the remainder of this paper. That is, we are concerned with the differences in the programming languages used to write FPs and the differences in hardware and software platforms on which the FPs are executed and how these differences affect the mutually cooperative execution of such FPs.

2.2 Approaches

In this section we will investigate three possible approaches to one aspect of the overall objective of mutually cooperative execution, namely the problem of

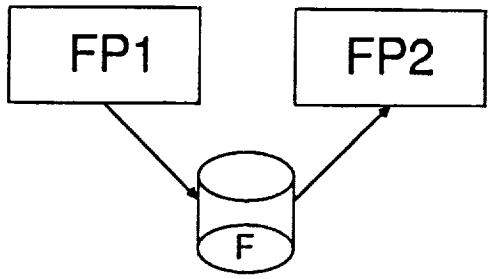


Figure 1: Operating System File

connecting FPs together. We have chosen three methods of such interconnection for further analysis: the *Piping Method*, which involves connecting programs together; the *Pairwise Language Interface Method*, which may be used to connect subprograms together; and the *Common Distributed Language Environment*, which encapsulates all FPs in a specification written in a common language.

2.2.1 Piping Method

A pipe is unidirectional conduit connecting two programs[6]. There are four distinct cases to be considered, depending on the level at which the connection is performed. In all cases we strictly consider a single pipe between two programs FP1 and FP2.

Common File Piping As shown in Fig. 1 this case involves connecting FP1 and FP2 through a file F. This requires that the operating system supply the file abstraction and the read and write connections to it. Performance is likely to suffer in this case, as without some method of synchronization FP2 must wait until FP1 has completely finished filling F before it may commence reading. In addition, if FP1 outputs a large amount of data to FP2, it is easily possible to overflow the limits of the file storage system.

This is the simplest type of pipe, since the operating system has the responsibility for managing the connection. The programs must still be rewritten to use them, however, and without synchronization there will obviously be some

[6]Although it is possible to connect two subprograms in this manner as well, it is usually more efficient to use a language interface, as described in Section 2.2.2.

Figure 2: Pipe

upper limit on the amount of data that may be written to the pipe, since it is, in effect, a buffer.

User Generated File Similarly, this case involves a file with read and write connections, but here the user must supply the mechanism. This must be done if the operating system does not supply the necessary primitives (in this case, input-output redirection) to obviate user rewriting of portions of each FP. Furthermore, this user-level rewriting must be performed for each pair or programs that are to be connected.

Here we have somewhat more control over the connection. The lack of synchronization exhibited by the previous case can be overcome by appropriately programming the interfaces to the file. For example, the file interface can be written to notify the operating system to suspend a producer if the amount of data in the file becomes large. However, the burden of maintaining the file abstraction now rests with the user.

Operating System Piping In this case we replace the file and associated read/write mechanisms with a *pipe* as shown in Fig. 2. Here FP1 writes data to one end of the pipe, and FP2 reads from the other end. The pipe is unidirectional. The major difference between a pipe and the previous file mechanism is that reads and writes are synchronized; the reader may begin reading immediately, and the pipe makes data available to the reader as soon as it is written to the pipe. This case requires the operating system to support the pipe abstraction.

This pipe abstraction allows concurrent operation of reader and writer, which is a significant advantage since it is no longer necessary for each stage of a pipeline built with these pipes to completely process its data before the next stage can begin processing.

User Generated Piping This case is similar to that of Fig. 2 in that a pipe is employed, except that it must be supplied by the user. This must be done if

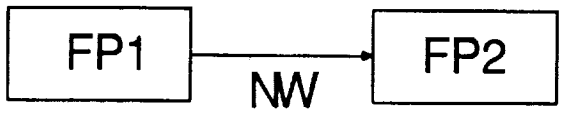


Figure 3: Network Pipe

the operating system does not supply a satisfactory piping function, and requires that the user re-write portions of each FP, again for each such pair of programs.

Correctly programming such a pipe abstraction at the user level is not a trivial task, particularly since any operating system not providing a pipe abstraction is not likely to provide the synchronization primitives necessary to implement one. Furthermore, such an implementation is destined to be non-portable.

Network Piping In the final case shown in Fig. 3 we again connect two programs via a pipe, but here the pipe involves communications across a network of some type. Implicit in the previous cases was the assumption that all programs were executing on a single processor, possibly under the control of a single operating system. Here the major differences are that two processors must be synchronized to use the pipe, accessing a network impacts the performance of the pipe, and the operating systems, languages, and processor architectures on which each program executes may be different.

If implemented correctly, a network pipe is a powerful abstraction because it is no longer necessary to remember whether the two ends of the pipe are located on the same processor. This makes writing software for distributed systems much easier. For example, the Berkeley UNIX socket abstraction can be used to create a pipe whose ends can alternately be located on the same machine, on two machines adjacent on the same local network, or on two machines separated by several thousands of miles; the software using the pipe does not have to be aware of the geographical dispersion in any way. The main problem is that transmission delays vary widely with many factors having nothing to do with the programs that use the pipe, and that while executing the same distributed program repeatedly will usually produce the same results, the same cannot be said for its performance.

In conclusion, we have examined five types of pipes as candidates for interconnecting programs. Since each pair of previously written programs must be rewritten if they are to use these pipes, we shall not consider them further in

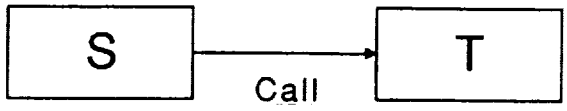


Figure 4: Pairwise Language Interface

this paper.

2.2.2 Pairwise Language Interface Method

Implicit in the Piping Method is the assumption that the connected FPs are programs which stand alone in whatever environment executes them. This forces all subprograms which make up such an FP to be written in the same language. We now relax this assumption by considering the interconnection of different subprograms within the same FP. Thus, we introduce the Pairwise Language Interface Method, a method that can be employed to connect subprograms.

In Fig. 4 we show the connection of subprograms S and T which are each written in different languages. In order to properly perform the call, the actual parameters specified by S must be translated to T's equivalent formal parameters[7]. This function is accomplished by a *call translation* function which must take place for each call. Any specific calling conventions expected by the called subprogram must be honored by the calling subprogram. This call translation must be discovered for each pair of languages whose subprograms are to be so connected, and an appropriate scheme for call translation for each such pair must be crafted.

In addition, the underlying run-time systems of the two subprograms, which might be different, must be accommodated. For FPs which have little or no interaction with their run-time systems, or whose run-time systems are independent of each other, this is not of great consequence. On the other hand, FPs whose run-time systems interfere with each other or which exhibit additional constraints, such as a typical Ada run-time system implementation which make it difficult for a non-Ada subprogram from invoking an Ada subprogram, will be more difficult to connect together. Again, pairs of such run-time systems must be investigated and points of interference resolved.

[7]And vice-versa, for result parameters.

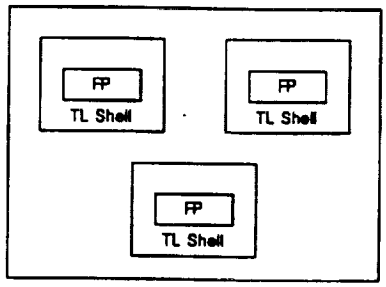


Figure 5: Common Distributed Language Environment

Since both call translation and run-time system interference require manual analysis and crafting, this approach is also not considered further.

2.2.3 Common Distributed Language Environment

Another way in which subprograms written in different foreign languages may be interconnected is to encapsulate them inside "shells" written in common *target language* TL. In Fig. 5 each subprogram FP may be written in a different *foreign language* FL, but since each is encased in a shell written in TL the interfacing of these subprograms is presumably simplified. Furthermore, since we now can write each program in the target language, connections between programs become simplified as well. Intuitively, the difficulty with this approach lies in the connection of an FP to its TL shell, because it is at that level that the issue raised in the Pairwise Language Interface method – call translation – must be addressed. This will be investigated in the next section.

3 Common Distributed Language Approach

In this section we investigate the Common Distributed Language Approach in detail. Since we have had previous experience with one particular distributed language – Distributed Ada – we first summarize its properties. After some preliminaries we then lay out the components of the approach.

The Common Distributed Language Approach is supported by the recent findings of the Defense Science Board [4]:

> Techniques for designing software in little modules, for defining

> the module interfaces precisely, and for using common file formats have come into standard use during the decade. These methods, the backbone of so-called "modern programming practices," radically improve the structure and adaptability of large programs.

The report goes on to state Ada's role in promulgating this improved adaptability:

> The Ada programming language is designed to make such modularization natural, and to provide very powerful facilities for linking modules. Integrated programming environments, such as Unix [sic], provide the same kind of facility at another level, that of the shell script linking whole programs together.

We observe that the Common Distributed Language Approach can be used to exploit Ada's natural facilities for linking modules together even when those modules are written in different languages, because of Ada's precision in defining module interfaces via compiled Ada specifications.

3.1 Distributed Ada

We have already explored the concept of a distributed language by developing Distributed Ada [11], which is based on the Ada [8] programming language. Ada was chosen as a base language because it possesses the following attributes (see, for example, Booch [3], Habermann and Perry [5], or Barnes [2]):

1. *Readability.* Since it is a recognized fact that most programs are read much more often than they are written, it is important to ensure that a program can be easily read and understood, particularly by those that did not originally write it. Ada strives to accomplish this through, among other things, a block-structured organization, a clear means of object qualification, and use of textual syntactic elements instead of special character sets.

2. *Object-oriented programming.* Rather than differentiate between components of the language, such as scalar variables, subprograms, constants, and so on, Ada operates in terms of *objects*, which greatly standardizes the specification and use of these components as objects without great regard for the component's true identity.

3. *Strong typing*. This forces each object to have a clearly specified set of values and forbids assignment or comparison of values not in the set to the object. This allows an Ada compiler to detect a large class of errors — operating on an object with unintended values — at compilation time.

4. *Data and program abstraction*. By hiding many implementation details from portions of an Ada program that do not need to know them, it becomes much easier to change these implementation details without unduly affecting the rest of the program.

5. *Separate compilation*. As an instance of abstraction, separate compilation splits the definition (the *specification*) of an Ada program unit from its implementation (the *body*). Other Ada program units refer only to the specification when dealing with the program unit; the body can be changed freely without interfering with the rest of the program.

6. *Libraries*. Ada program units can be declared to be *library units*. As such they can be referenced by many different programs using a single source-level *with* statement. This facilitates bottom-up software design at the language level as has been investigated by Rajlich [9][10].

7. *Exception handling*. Ada provides explicit means to handle unforeseen or exceptional conditions in a program. The exception handling mechanism can be layered to design programs that contain the effects of these conditions without affecting the rest of the program.

8. *Tasking*. The Ada language supports the concept of concurrent programming through the use of language-level context objects called *tasks*. Ada tasks are program units which execute independently of other program units unless they choose to interact with other tasks via a *rendezvous*, which resembles a subprogram call.

9. *Generic units*. In order to avoid repetitive programming, the Ada language allows the definition of *generic units*, which are essentially templates for creating program units which are independent of the types of objects they manipulate. Copies of this template are then created (*instantiated*) as needed by the Ada program for specific object types from this generic template. These object types are not limited to data objects; a user-defined

subprogram, for example, can serve as such a *generic formal parameter* when instantiating a generic unit.

Based on the Ada programming language, we have defined Distributed Ada to explore the concept of a distributed language while taking full advantage of the above attributes. Briefly, the distributed language concept differs from traditional methods of programming a distributed system in that the programmer writes a single program which is mechanically decomposed – via embedded distribution directives – into a set of programs which can then be separately compiled and installed on each machine in the distributed system. This decomposition is called pretranslation, and is performed by our Distributed Ada Pretranslator. This methodology has the following advantages:

- A single program is written; consequently, the additional complexity usually induced by requiring the writing of a separate program fragment for each of the processors in the distributed system is avoided.
- The powerful automatic error detection techniques available in modern languages are extended across processor boundaries. This is particularly useful for the development of large systems.
- There is no loss of abstraction comparable to that which occurs at the interfaces between separate program fragments. Since a single program is written, it is straightforward to ensure consistency between its components.
- The pretranslator output consists of a set of Ada programs which may be compiled by existing Ada compilers, thus no compiler modifications are required.

The advantages of a distributed language should now be clear. It may still be argued that another common distributed language might be suitable instead of Distributed Ada. We reject this argument on the basis of Ada's obvious strengths in modularization via packages and its facilities for linking separate modules; these conclusions are supported by the Defense Science Board's conclusions quoted above.

For these reasons we have chosen Distributed Ada as our target language.

3.2 Related Work

Several recent efforts in the development of tools and methodologies for the conversion of programs written in one language into another and for the connection of existing programs into a single cooperative program object are summarized here.

Wallis [12] examines various methodologies for the automatic conversion to Ada of software written in other languages. He observes that it is not sufficient to translate a program by replacing constructs with equivalent constructs in a new language, but that these new constructs must imply a change in how one reasons about the program and the process of creating it. He then concludes that "for both managerial and technical reasons, automatic language conversion cannot provide an easy route to a complete conversion to Ada." However, he advocates the use of converters as a semi-automatic "bridging technology" when it is desired to convert a body of existing code to Ada.

Kernighan [6] describes an archetypical solution to the problem of converting programs from RATFOR – a "rational" FORTRAN dialect – to FORTRAN: the use of a preprocessor that accepts RATFOR and produces equivalent FORTRAN, which is then compiled with a standard FORTRAN compiler. The approach differs from the previous case because it is clearly never intended that a conversion from RATFOR to FORTRAN be made; rather, this method allows the programmer to write RATFOR code and never consider the equivalent FORTRAN.

McMillan et al [7] provide a tool which assists in the serial to parallel conversion of FORTRAN programs. They have constructed an interface recovery system which utilizes various existing tools such as compiler listing generators and cross-reference utilities to create a global data structure indicating read/write variable accesses by the subroutines of a typical large FORTRAN program. This table can then be used as an aid in transforming this large program into one that would execute on a multiprocessor with optimal placement of these variables. The relevance of this work is in its usage of existing tools to analyze existing programs, and in its use of the "workbench approach", that is, the construction of tools to aid in the solution of a potentially difficult problem when it appears that a fully automated solution is not feasible.

Our approach differs from and complements these methods in the following ways:

- We do not advocate the wholesale translation of programs from another language to Ada. We do wish to utilize these programs, but without

source-level translation and preferably without a great deal of source-level inspection.

- One function performed by our pretranslator is similar to the RATFOR preprocessor, and we would like to maintain this abstraction for Distributed Ada.

3.3 Objective

In light of the discussion in the previous section, we can restate our objective – the development of techniques which allow programs to function together – into more precise terms. The objective now concerns the transformation of existing software systems into Distributed Ada. This can involve two cases:

- Transforming distributed program fragments written in some language and executing on some processor and operating system configuration into a Distributed Ada program.

- Transforming a set of unrelated programs written in possibly different languages and running on possibly different processor and operating system configurations into a Distributed Ada program.

The difference here is that in the first case the original program was written as a single, distributed program, and the problem is to transform the program into a Distributed Ada Program, while in the second case the problem is to transform pieces of previously written and therefore unrelated programs into a single Distributed Ada program.

It is important that these transformations be performed with minimal modifications to the original programs.

3.4 Strategy

Recall that a program written in an existing language is called a *foreign program*, abbreviated FP. Let us now specify Distributed Ada as our target language TL, and define a program written in Distributed Ada as a *target program*, abbreviated TP. Furthermore, foreign and target subprograms will be abbreviated FS and TS, respectively. The process of changing a foreign program or set of programs into a target program is called a *transformation*.

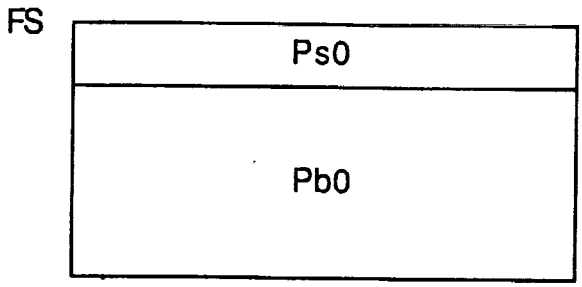


Figure 6: A Foreign Subprogram

The general strategy will be to encapsulate each FS in a shell written in TL; the resulting subprogram, now a TS, will then represent a program unit equivalent in functionality to the original FS. Collections of these TSs are then available for the construction of new TPs, facilitating the re-use of these TSs.

In general we will concern ourselves with the transformation of FPs containing subprograms to equivalent TPs. Although FPs could contain task-like constructs that might be transformed into Ada tasking constructs, we will not consider this type of transformation in this paper.

Thus, within a given FP or TP there will reside a number of subprograms which incorporate the functionality of the enclosing program. The specifications[8] of these subprograms are denoted *Ps*; the bodies are denoted *Pb*. FPs lacking specification parts elide *Ps*.

FPs and TPs execute in – probably different – *environments* provided by their respective host operating and run-time systems. For the purposes of this paper, such *underlying system* (US) services requested by subprograms executing in a given FP or TP are modeled as calls on *external* subprograms.

3.5 Basic Transformation

In this section we discuss the mechanics of the basic transformation of a given FS to TS by considering FS alone. The next section will then examine the more difficult problems concerning FS interaction with its US environment.

[8]In Ada, a *specification* of a program unit describes the interface (number and types or arguments, etc.) to the program unit; a corresponding *body* describes the implementation of the program unit.

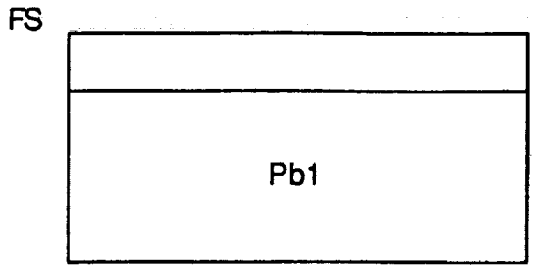


Figure 7: A Body-Only Foreign Subprogram

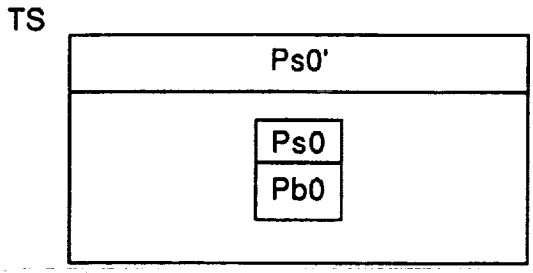


Figure 8: Transformed Foreign Subprogram

Figure 6 shows a typical FS split into two sections: the upper portion is the specification section, and the lower portion is the body. Depending on the nature of FL a specification may or may not exist for FS. In Fig. 6, P_0 has both a specification and a body, which are denoted Ps_0 and Pb_0 respectively. The other case is shown in Fig. 7 where P_1 has only a body, denoted Pb_1.

Figure 8 shows the transformation required by our strategy. Here Ps_0 has been replaced by Ps_0' and the body Pb_0 has been encapsulated by a *shell* of TL software. Figure 9 shows the case where the foreign program has only a body; a Ps_1' has been generated, and again a target software shell encapsulates the body Pb_1.

The interface between TS and FS is constructed as follows. There are two cases to be considered, controlled by the presence or absence of a specification in FS. If the specification is present, we generate a TS specification Ps_0' and an *interface procedure* I_0 as shown in Fig. 10. The body of I_0 consists of a procedure which makes a direct call to Pb_0 (via Ps_0). The direct call is set up

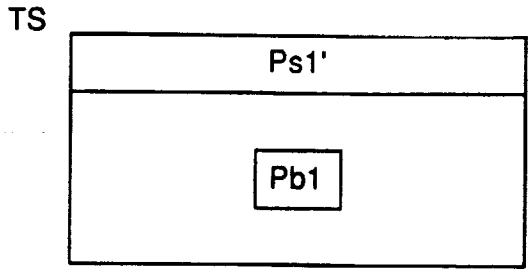


Figure 9: Transformed Body-Only Foreign Subprogram

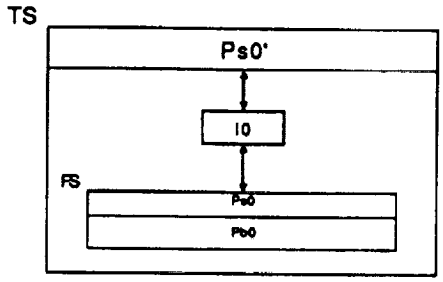


Figure 10: Interface Procedure I_0

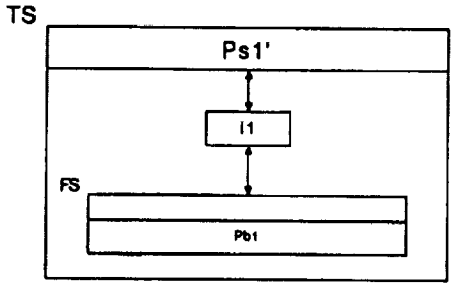


Figure 11: Interface procedure I_1

by declaring Ps_0 a **pragma INTERFACE** procedure in I_0. In the second case, when a specification is absent in FS, the same methodology is used, except that in Fig. 11 interface procedure I_1 is generated by examination of the body Pb_1.

We distinguish between these two cases because we wish to make clear that the absence of a specification in the latter case will make the transformation more difficult to construct. For example, in the absence of a specification, constraint information cannot be determined without a detailed examination of the body for any dynamic checks on the bounds of a particular variable.

As an example of the preceding process, consider the following FS written in the FL FORTRAN:

```
SUBROUTINE INVERT(A,N,RC)
DIMENSION A(1,1)
INTEGER RC
...
```

Here **INVERT** inverts a square matrix **A** of dimension **N**. The success or failure of the inversion operation is returned in **RC**. We wish to encapsulate **INVERT** in a TL shell. The transformation is then:

```
type MATRIX is array (INTEGER range <>,
  INTEGER range <>) of REAL;
pragma INTERFACE(FORTRAN,INVERT);
procedure TINVERT(A: in out ARRAY; N, RC: INTEGER) is
begin
  if A'LENGTH(1) /= A'LENGTH(2) then
    raise NONSQUARE;
  end if;
  INVERT(A,N,RC);
end INVERT;
```

Here **TINVERT** is the name of the new TS which encapsulates FS. The body of **TINVERT** (that is, I_0) first performs a check to ensure that a square matrix has been passed, and then calls the FS **INVERT**. The results of the inversion are then returned to the caller of **TINVERT**.

3.6 More Complex Transformations

The previous simple transformation suffices to make libraries of foreign subprograms available for reuse in new Ada programs. We would now like to explore some more complex problems which do not obey the simple library paradigm. At the same time, we expand the scope of our consideration to that of target programs, that is, programs composed of one or more TSs.

Therefore, in this section we analyze two kinds of more complex transformations. The first kind allows for a richer interface to a given FS than that provided by the simple call-return interface above; this kind of transformation allows calls made by a given TS within a TP to subprograms external to the TP. This complements the basic transformation of the previous section by specifying ways in which foreign code embedded in a given TP can call target code.

The second kind of transformation allows the interconnection of TPs generated from FPs that may or may not have been designed for such interconnection.

3.6.1 Calls from TP

In the previous section we have examined a basic transformation which allows calls from subprograms external to a given TP to call foreign subprograms encapsulated in TL shells contained in the TP. The inverse situation, calls from encapsulated FSs in TP to subprograms external to TP, is more difficult. The reason for this is that the embedded bodies Pb_0 – which are *foreign* language subprograms – are required to call subprograms specified in this or another TP. However, this results in foreign code calling target code subprograms. This linkage is not uniformly supported by any Ada compiler because such linkage is not mandated by the Ada standard; those compilers that do support it, such as the compiler supplied by the Verdix Ada Development System (see below) do so in a very implementation- and compilation-dependent manner, since no standard exists for this type of call.

We have identified two ways in which the foreign-to-target linkage may be accomplished: employ a direct foreign-to-target call utilizing an implementation that must be crafted for each target compiler, or devise a method in which such calls are made – indirectly – via target-to-foreign calls.

Foreign-to-Target Calls As shown in Fig. 12, we have an embedded FS which desires to call TS P.

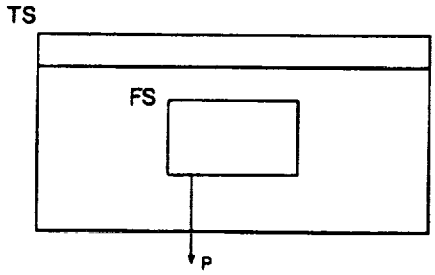


Figure 12: Foreign-to-Target Call

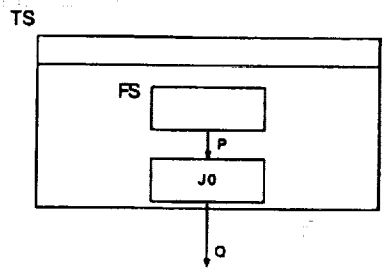


Figure 13: Interface Procedure J_0

In Fig. 13 we show how this linkage may be accomplished. Within TP we provide a new interface procedure J_0, written in FL, which must be designed to accept calls by subprograms within FP originally destined for P. J_0 completes the linkage by invoking TS procedure P directly via its compiler-derived name Q.

The result of these transformations is that FSs need not be changed. Rather, $J0$ manages the differences between the FS names invoked by FS code and any compiler-induced changes to those names that occurred when the invoked FSs were encapsulated by TL shells. The drawback is in the nature of the linkage between the calls on P in FP and J_0: these are foreign-to-target calls, and the implementation of J_0, that is, the linking of an invoked FS and J_0, must be crafted for each differing target compiler implementation. In some cases, the compilation topology influences the names that are generated for TS names; this presents an additional complication.

As an example, consider one compiler which supports this type of linkage, namely, the compiler provided by the Verdix Ada Development System. The Ada compiler supplied with this system permits Ada subprograms to be invoked by foreign subprograms; however, the name by which the Ada subprogram must be referenced by the foreign subprogram is derived by the Verdix Ada compiler from the declared name and from the surrounding program structure. For instance, a foreign subprogram wishing to invoke an Ada subprogram `P` embedded in an outer procedure `MAIN` must specify a name like `_A_p.2B13.main`. Not only can this derived name change from one compilation to the next if the program structure in which the subprogram declaration is embedded is altered, but it is unlikely that another compiler vendor will apply the same derivation rules.

In Fig. 13, assume that FP is a program written in the C programming language which wishes to invoke Ada subprogram `P` as above; it wishes to do this because `P` is a TL shell encapsulating a FL subprogram `P` (perhaps also written in C) whose services were employed by FP prior to its encapsulation. $J0$ then will be a C function named `P`, and it will call `_A_p.2B13.main`:

```
J0()
{
    _A_p.2B13.main();
}
```

A problem may arise here, because many C compilers will not accept `_A_p.2B13.main` as a valid function name because of the embedded peri-

ods. In this case, it will be necessary to replace the periods with some legal character, say the letter `X`, and then to alter the object program output to replace the periods in the external symbol references. Clearly, this is not very satisfactory, but this example does illustrate the contortions that may be required in order to craft a solution based on a particular compiler vendor's transformation rules and the rules describing FL[9].

Target-to-Foreign Calls One solution to the problem of crafting compiler-specific foreign-to-target procedure calls is to replace them with target-to-foreign calls given a satisfactory methodology for doing so. The most obvious way to accomplish this is to devise a polling scheme. For example, assume procedure R in FP desires to call procedure P in TP. In this case, P may periodically call R until this latter procedure indicates it is ready to communicate with P. It should be clear that this is a very unsatisfactory method for supporting target-to-foreign calls because of its inherent polling cost.

Another approach which avoids this cost employs a foreign-to-target interrupt followed by a target-to-foreign procedure call. This is a promising approach because the TS need not spend time performing a potentially wasteful polling operation. However, this method requires that a FS be able to cause a TP interrupt entry to execute, which again requires some specific crafting which depends on the compiler and run-time system being used. The advantage here over the previous case where FS name mappings had to be encoded into $J0$ is that once a method is discovered to cause a given interrupt entry to execute, changes in the structure of TS will most likely not require a new method, whereas in the previous case the compiler-generated name could change, requiring additional steps to discover the new name.

The basic mechanism is illustrated below. Suppose foreign subprogram FS wishes to call target subprogram TS. A target program TP may be constructed as follows:

```
package TP;
package body TP is
  task type T is
```

[9]In fact, it may be noted that if such editing of object module external symbol references is permitted, then it would be simpler in this case to declare $J0$ as being defined externally, to omit the body of $J0$ entirely, and to replace the resulting external symbol reference $J0$ with `A_p.2B13.main`.

```
    entry E;
  end T;
  type TA_T is access T;
  TA: TA_T := new T;
  task body T is
  begin
    loop
      accept E;
      TS;
    end loop;
  end T;
begin
  RTS_LINK(TA,1,15);
end TP;
```

Here **TA** is an access (pointer) to an instance of the task **T**. We assume **TP** and **FP** are served by an Ada run-time system and are encapsulated by a single UNIX process, and that the **RTS_LINK** call is provided by the compiler vendor as a means for linking UNIX signal number 15 with the first entry of task **T**, namely **E**[10]. Once **RTS_LINK** is called, any receipt of signal number 15 by the UNIX process encapsulating **TP** will result in the completion of the **accept** statement and the subsequent execution of subprogram **TS**.

The foreign subprogram **FS** may now cause **TS** to be executed by:

```
FS()
{
    kill(getpid(),15);
}
```

Here **FS** sends signal number 15 to the UNIX process encapsulating it (the identity of the encapsulating process is obtained via **getpid()**); since this process also encapsulates **TP**, the subprogram **TS** is invoked via the mechanism described above.

A potential drawback to this method revolves around the necessity of sending a signal to a UNIX process in order to initiate the foreign-to-target call, and

[10]Alsys, another Ada compiler vendor, provides a method very similar to the one described here for attaching Ada task entries to DOS interrupt vectors.

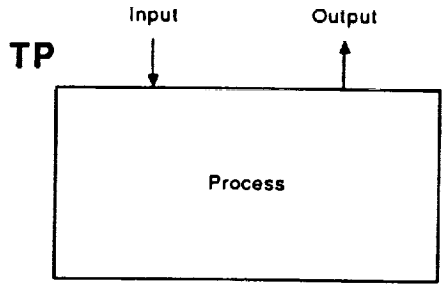


Figure 14: TP Model

the subsequent scheduling of task T. Both of these operations can be expensive, although it may be argued that direct polling as described earlier is more expensive, depending on the frequency with which it is necessary to poll and the rate at which the foreign-to-target calls are made.

3.6.2 Interconnecting TPs

Concepts In this section we reconsider the problem of connecting TPs together via a pipe, as discussed previously in Section 2.2.1. In that section we were concerned with connecting FPs together, and concluded that the rewriting of FP code necessary to do this made that method less attractive. In this section, we will re-examine this problem for TPs, that is, after the methods of the previous section have been applied to encapsulate each FP in a TL shell; in particular, we wish to examine the effects of this encapsulation on the performance of interconnected TPs. As before for FPs, we will be concerned with the interconnection of TPs that may or may not have been designed for such interconnection. In addition, we will restrict the class of programs to those employing the *input-process-output* model (described below).

As shown in Fig. 14, we model each TP that we wish to connect via a pipe as a program separated into three distinct phases:[11]

- **Input.** Accept parameter variables from the process input.
- **Process.** Perform the embedded FPs intended function.
- **Output.** Deliver result variables to the process output.

[11]TPs whose functionality cannot be separated into these phases are not considered.

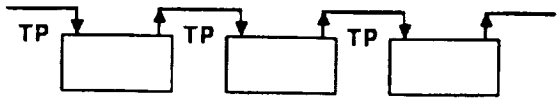


Figure 15: TP Pipeline

In the sequel we shall refer to parameter and result variables as *objects*.

Several of these TPs may be connected into a pipeline as shown in Fig. 15, where the input of a TP comes from the previous TP and the output goes to the following TP.

A problem occurs when connecting TPs containing FPs that were not designed to interact, for then there is no guarantee that the objects produced by one stage of the pipeline will be syntactically compatible with the input requirements of the next. Some kind of conversion must then be performed between the two, and associated with this conversion is a conversion cost; it is of vital importance to reduce this performance cost to a minimum. There is a generality/performance tradeoff which must be taken into account, similar to that which occurs in inhomogeneous networks such as Bolt, Baranek, and Newman's Cronus distributed operating system. The problem is the following.

Theoretically, only one syntax converter is needed between the output of one stage of the pipeline and the input of the next, and it can be placed either within the sending TP or the receiving TP. Consider the former case. An output format must be chosen for the converter; the logical choice is the input format required by the successor stage. But in fixing the output format in this fashion, we are limiting the use of the output to precisely those successor stages that understand the chosen output format. If we wish to connect a TP which expects a different format we must change the converter in the sending TP, something we do not wish to do because we shall have to generate a different version of the sending TP for each new receiving TP, or build a new TP out of the FP and the appropriate output converter when we wish to set up the pipeline. For static, long-lived pipelines this may be adequate, but it will not do if we wish to provide these TPs as a library for general use. A similar argument may be given for input converters.

The other alternative is to provide input and output converters for each TP, and fix a standard format to which all output converters must convert and from which all input converters must convert. This doubles the number of conversions

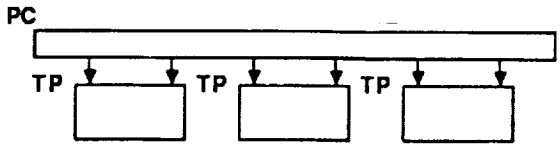


Figure 16: Pipeline Controller

that must be performed.

For maximal generality the latter method is preferred, as no new converters must be written when new stages are added to a pipeline. For performance, however, the number of conversions must be minimized, at the cost of some rewriting of converters whenever the pipeline must be rearranged. The requirements of the application will usually force one or the other methodology. Since we are more concerned with minimizing the rewriting of code and the generation of reusable libraries, we choose to examine the latter strategy, and fix a standard format called the *network format.*

Conceptually then, input and output conversions are themselves composed of two conversions. Consider the problem of passing input objects to a TP. This entails converting the object from network to TP format; let this conversion be done by an interface procedure T_N^T. We must still convert the object from TP format to FP format. This is done by procedure I_0 in Fig. 10; this part of I_0 is herewith renamed T_T^F. There are, of course, the comparable subprograms T_T^N and T_F^T required to pass an object from FP to the next stage in the pipeline. In practice, we can collapse the four conversions into two in the obvious way:

$$T_N^T \circ T_T^F \rightarrow T_N^F$$

$$T_F^T \circ T_T^N \rightarrow T_F^N$$

It must be noted that T_N^F and T_F^N must be crafted for each FP.

Pipeline Controller Conceptually, the foregoing allows us to connect an arbitrary number of foreign code modules into a pipeline. In practice, the target-to-foreign call problem occurs when a TP attempts to pass an object to the input of the next TP. This problem can be solved by the methods of Section 3.6.1, or we can utilize a *pipeline controller.*

In [1] Antonelli et al devised a method of generating hierarchical executable descriptions of manufacturing cells. Each level of the hierarchy contained a

control procedure which called the other subprograms at that level of abstraction in the proper sequence. In utilizing this methodology, we propose a pipeline controller PC which replaces the control portion of the pipeline itself. In Fig. 16, the PC – which is written in the target language – makes calls on the various TPs as required to implement the semantics of the pipeline. This removes both the need for hand-crafting foreign-to-target calling sequences and relying on polled target-to-foreign calls. It is important to understand that all calls in Fig. 16 are from the PC to the pipeline stages; the first call to a stage passes the input objects, and the second call to a stage retrieves the output objects. If the performance penalties inherent in such serialization are to be avoided, some sort of multitasking must occur in the PC.

4 Conclusion

We have discussed some of the issues involved in transforming foreign code packages into cooperative target programs. After an examination of several other approaches, a methodology based on encapsulating shells of a common target language was selected. We selected Distributed Adaas the target language of choice, outlined several approaches in connecting encapsulated foreign programs together, and presented solutions to several problems with these approaches.

Possible future work would include an investigation into the more difficult problem of transforming arbitrary independent foreign code packages into a single, distributed target package, and an investigation of the role of tools, such as call-graph analyzers and skeleton generators, as aids in the transformation process.

Bibliography

[1] Charles J. Antonelli, Richard A. Volz, and Trevor N. Mudge. Hierarchical decomposition and simulation of manufacturing cells using Ada. *Simulation*, 46(4):141–152, April 1986.

[2] J. G. P. Barnes. *Programming in Ada.* Addison-Wesley, second edition, 1984.

[3] Grady Booch. *Software Engineering with Ada.* Benjamin/Cummings, 1983.

[4] Defense Science Board Task Force. Report of the defense science board on military software. Technical report, September 1987.

[5] A. Nico Habermann and Dewayne E. Perry. *Ada for Experienced Programmers*. Addison-Wesley, 1983.

[6] Brian W. Kernighan. RATFOR - a preprocessor for a rational Fortran. *Software – Practice and Experience*, October 1975.

[7] Donald J. McMillan and Alan W. Baum. An interface recovery system for serial-to-parallel conversion of Fortran 77 programs. Technical Report GMR-5889, General Motors Research Laboratories, Warren, MI 48090, July 1987.

[8] The United States Department of Defense. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983 edition, February 1983.

[9] Vaclav Rajlich. Refinement methodology for design with packages. Computer Research Laboratory Technical Report CRL-TR-31-83, The University of Michigan, 1983.

[10] Vaclav Rajlich. Paradigms for design and implementation in ada. Computer Research Laboratory Technical Report CRL-TR-43-84, The University of Michigan, 1984.

[11] R. A. Volz, P. Krishnan, and R. Theriault. An approach to distributed execution of Ada programs. In *JPL Workshop on Ada Programs*, Jan 20-22 1987.

[12] P. J. L. Wallis. Automatic language conversion and its place in the transition to Ada. *Ada Letters*, V(2):275–284, September 1985.