MODULE REUSE BY INTERFACE ADAPTATION

James M. Purtilo Joanne M. Atlee

SUMMARY

This paper describes a language called NIMBLE that allows designers to declare how the actual parameters in a procedure call are to be transformed at run time. Normally, programmers must edit an application's source in order to adapt it for reuse in some new context where the interfaces fail to match exactly (e.g., the parameters may appear in a different order, data types may not exactly match, and some data may need to be either initialized or masked out when the reusable module is integrated within a new application.) But NIMBLE allows programmers to adapt the interfaces of existing software without having to operate on the source manually. As a result, existing software may be easily reused in a broader range of applications, and software libraries do not need to store many variants of a component that differ only in how the interfaces are used. NIMBLE has been implemented on a variety of Unix hosts, and is part of a broader reuse project at the University of Maryland. Our current system is suitable for use either in conjunction with existing module interconnection languages, or stand-alone with C, Pascal and Ada source programs.

Key words: module interconnection language, software reuse.

Complete mailing address:

Professor James Purtilo
Computer Science Department,
University of Maryland
College Park, Maryland 20742

INTERNET: purtilo@cs.umd.edu **PHONE**: 301 405 2706

INTRODUCTION

The ability to reuse components is an economic necessity within software development projects, and is critical within prototyping efforts. Systems to help identify reusable components are increasingly able to suggest application modules with the specified functionality [1]. However, often such modules must be adapted in some manner before they can be used. For example, parameters may appear in a different order, data types may not exactly match, and some data may need to be either initialized or masked out when the reusable module is integrated within a new application. As the amount of adaptation increases, the economic benefit of reusing that component is decreased.

In order to reduce the cost of adapting software for reuse, we turn to automatic techniques. Typically, the emphasis has been upon transformation of abstract specifications into a valid implementation, where tailoring of interfaces is performed during generation of the source programs, such as shown in References 2 and 3. A more recent development is a language for transforming control structures of an Ada-like language [4]; in this way programmers can easily adapt algorithm implementations for application to elaborate data sets, and tailoring of interfaces is again performed during the source to source transformation.

To complement the above techniques, we have developed a language called NIMBLE for programmers to declare how actual parameters of a procedure invocation should be transformed at run-time. This approach keeps the source programs of both components intact, and focuses on coercing parameters instead. Programmers use NIMBLE to declare a map from the calling program's actual parameter list to the formal parameter list of the procedure being invoked. Then our system transforms the user's map into an execution-time module that performs the desired coercion at each invocation.

NIMBLE is oriented for use in the Polylith software interconnection system, a "software bus system" [5]. The combination of these two systems affords us a powerful resource for developing application structures, and then reusing previously implemented modules to execute those designs. However, NIMBLE can also be used stand-alone, and results are applicable to any software development environment that supports separate compilation and packaging of reusable modules.

THE CASE FOR INTERFACE ADAPTATION

There are several reasons for programmers to consider adaptation of interfaces alone, not the least of which is when only the object code is available for an existing component. In order to illustrate these points, consider the example in Figure 1. The module on the left contains a set of employee records, where each record holds pertinent data concerning one employee. This module makes a procedure call to PRINTLABEL, defined on the right, which is a reusable envelope-printing routine. This second module requires an interface consisting of only the sex, name, and address of the employee.

These components cannot be linked directly due to differences in their respective interface patterns. Not only does the record structure expected by the called module contain fewer fields, but the order of the fields has changed and the sex data field is now represented by an integer. Clearly some data manipulation is required.

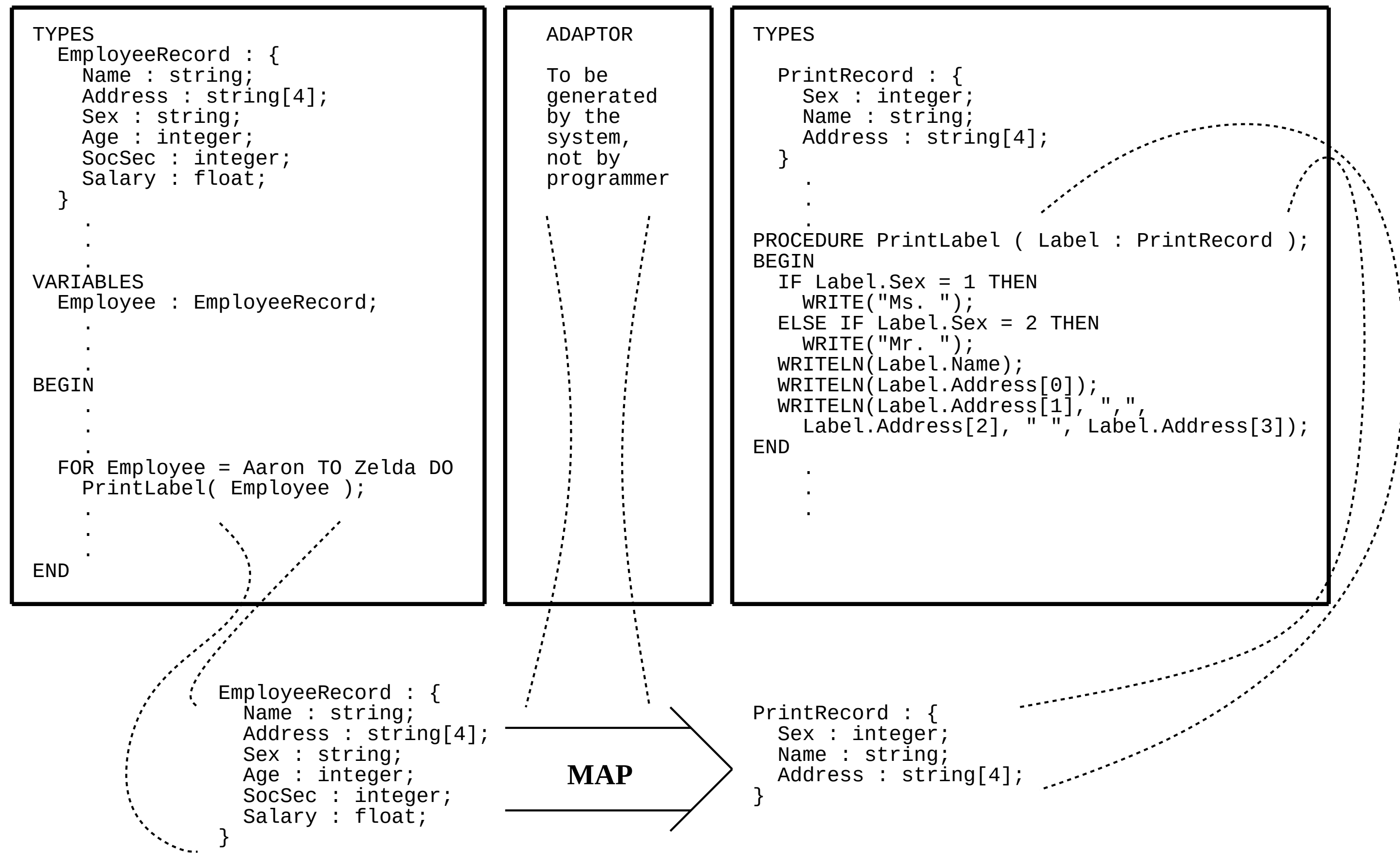
In order to interconnect these two software components, the programmer could rewrite one of the modules to meet the interface of the other. But he could also adapt the interface externally, by either

- introducing a separate procedure, to catch the original call to PRINTLABEL with one set of parameters, then make the actual call to it using the rearranged parameters; or
- changing the code generator so it will build the PRINTLABEL activation record with parameters in the rearranged order at run-time.

In either case, the programmer needs some easy way to express how the parameters should be manipulated, and this is the purpose of our language NIMBLE, as will be described. NIMBLE focuses upon the first approach, that is, transforming parameters at run-time using a separate component linked into the application. While introducing the potential for a minor performance loss in the form of an extra procedure call, this approach does not require alteration of any of the user's language compilation tools.

In the context of our example problem, we now review the principle reasons to consider external, automatically-generated interface adaptation.

- **No source available.** When the two components from Figure 1 are only available in object form, then external adaptation of interfaces is the only option open to programmers. Often programmers write the interfacing code manually. In our approach, the coercion of an interface structure may be expressed abstractly, with the code to implement the coercion generated automatically. We can easily envision how this situation might be exploited by software vendors who are concerned with protecting intellectual property: they may deliver reusable software components in object form, along with an interface adaptation resource to give customers greater leverage in using the product set.
- **Simpler source programs for remaining components.** When source code is available for one of the components from Figure 1, then currently most programmers would manually adapt it to match the interface of the other component. But external adaptation may be valuable even in this case. Introducing extra interfacing code into an existing component makes it more complex and more costly to maintain. For example, Figure 2 illustrates how one of the routines from Figure 1 would grow in complexity just to have the interface structures correspond. In our approach, such code would be generated automatically, allowing the application source to remain simpler and easier to maintain — fewer non-functional constraints affect the application code.



```
TYPES
  EmployeeRecord : {
    Name : string;
    Address : string[4];
    Sex : string;
    Age : integer;
    SocSec : integer;
    Salary : float;
  }
    .
    .
    .
VARIABLES
  Employee : EmployeeRecord;
    .
    .
    .
BEGIN
    .
    .
    .
  FOR Employee = Aaron TO Zelda DO
    PrintLabel( Employee );
    .
    .
    .
END
```

```
TYPES

  PrintRecord : {
    Sex : integer;
    Name : string;
    Address : string[4];
  }
    .
    .
    .
PROCEDURE PrintLabel ( Label : PrintRecord );
BEGIN
  IF Label.Sex = 1 THEN
    WRITE("Ms. ");
  ELSE IF Label.Sex = 2 THEN
    WRITE("Mr. ");
  WRITELN(Label.Name);
  WRITELN(Label.Address[0]);
  WRITELN(Label.Address[1], ",",
    Label.Address[2], " ", Label.Address[3]);
END
    .
    .
    .
```

```
EmployeeRecord : {
  Name : string;
  Address : string[4];
  Sex : string;
  Age : integer;
  SocSec : integer;
  Salary : float;
}
```

```
PrintRecord : {
  Sex : integer;
  Name : string;
  Address : string[4];
}
```

Figure 1: Interface patterns to be coerced at run-time.

```
TYPES
  EmployeeRecord : {
    Name : string;
    Address : string[4];
    Sex : string;
    Age : integer;
    SocSec : integer;
    Salary : float;
  }
  PrintRecord : {
    Sex : integer;
    Name : string;
    Address : string[4];
  }

VARIABLES
  Employee : EmployeeRecord;
  Temp : PrintRecord;
  I : Integer;
```

```
BEGIN
  FOR Employee = Aaron TO Zelda DO
    BEGIN
      Temp.Name = Employee.Name;
      FOR I = 0 THRU 3 DO
        Temp.Address[I] = Employee.Address[I];
      IF STRINGEQUAL("male",Employee.Sex)
        THEN Temp.Sex = 1; ELSE
      IF STRINGEQUAL("female",Employee.Sex)
        THEN Temp.Sex = 2; ELSE
      Temp.Sex = 3;
    PrintLabel( Temp );
  END
END
```

Figure 2: Example of a manually-adapted component.

```
NIMBLE:     { EVAL( Usermap, Sex) ; Name ; Address }
```

Figure 3: Nimble declaration for the introductory example.

```
Usermap( s )
char *s;
{
     if (strcmp( s , "male" ) == 0) {
          return( 1 );
     } else if (strcmp( s, "female" ) == 0) {
          return( 2 );
     } else return( 3 );
}
```

Figure 4: Sample implementation of Usermap.

- **Reduced opportunity to introduce error.** Even when source is available for all application components, external adaptation has possible advantages. Whenever programmers must manually adapt a module for reuse, there is a good chance they will break it. A single declaration concerning rearrangement of parameters is easier to construct than are the changes to a (possibly large) application component. (While our notation must still be explained, the NIMBLE map to adapt Figure 1's interfaces is shown in Figure 3.) The resulting simplification of remaining components may also make them easier to reason about by automatic techniques.

- **Reduced complexity for underlying configuration management (CM) system.** Realistic programs often have many data that are similar in their use, but which differ slightly in structure for reasons of convenience or performance. When the implementation language does not support inheritance, then the programmer is faced with implementing many similar variations of essentially the same functions to operate on all variant structures. Implementation cost aside, managing these variants in the configuration can be expensive over the life of the project. In our example, the 'envelop printing operation' (implemented on the right in Figure 1) may need to be performed for many types of records in the database, each having the necessary information but laid out in different structures. (In other words, there may be many routines such as on the left in Figure 1, each making the call to PRINTLABEL() with a different record type.) With external interface adaptation, the programmer only needs to create one abstract map to show correspondence between each pair of records.

  There is an additional consideration. Before software components are baselined in a configuration management system, they often undergo a costly validation activity. Once they are validated, each change to the baseline requires revalidation, which is just as costly; this is true even if the change is to adapt the interface for use in a new application. To avoid paying this cost, developers may elect to leave the component intact within the CM system, and use external adaptation instead.

- **Prototyping.** Prototyping is an experimental activity intended to expose essential properties of an application before design decisions become irreversible. The cost of creating prototyping apparatus must be significantly less than the cost of creating the product, or else developers will gamble — the first version of the product is, in effect, the prototype. Therefore, to keep prototyping costs down, reuse of software is critical. External adaptation, based on abstract declarations, allows programmers to drawn upon a broader range of existing components rapidly, and minimizes the amount of interfacing code that must be written manually in order to construct the prototype.

The above list provides motivation for why a programmer would consider automatic, external interface adaptation. But any time extra code is introduced into an application, there is the question of what run-time performance cost must be paid. Our research has been to implement a notation for declaring interface maps, create a translator to generate the interface codes based upon those maps, and then study the effect of introducing such maps. After describing the language itself, we will present both the context of use and our experiences with this system.

A NOTATION FOR ADAPTATING INTERFACES

Nimble is a declarative language whose tokens correspond to the actual parameters of a program's interface (at the point of a procedure call). Its sentences express how those parameters should be rearranged and transformed in order to match the formal parameters of the interface (where the procedure is defined).

The first step is for the programmer to create the desired map, as described below. Then this map is compiled and used to generate an implementation of the map, called an adaptor. Finally, the adaptor is integrated into the application system, as will be discussed.

COERCION OF INTERFACE PATTERNS

The programmer's task is to declare a Nimble map based upon the pattern of parameters in both the formal and actual interfaces, and so the first step is to isolate the type structure of these interfaces. One way programmers can obtain these patterns is to simply transcribe them from the source. However, programmers can be notoriously poor translators, so we usually employ automatic techniques for extracting the interface patterns. Systems such as NewYacc can easily extract just this information from source codes [6], and this is what we provide in Nimble. If source codes are unavailable, or if the programmers use Nimble in an application language for which no interface extractor exists, then the programmers must reproduce their own interface descriptions from program documentation. Regardless, it is clear that the patterns must directly match the interfaces as they appear in the application source code.

In addition, the actual parameters — at the point of the call — must be given in terms of an *annotated* interface pattern, that is, each parameter is assigned a unique name. This labeling allows the programmer to refer to parameters individually when constructing a map from the actual list of parameters to the formal interface pattern.

For example, the two sentences provided to the programmer for our introductory example are

```
ACTUAL PATTERN:
Name:STRING; Address:STRING[4]; Sex:STRING; Age:INTEGER; SocNum:INTEGER; Salary:FLOAT

FORMAL PATTERN:
Sex:INTEGER; Name:STRING; Address:STRING[4]
```

In this case, we have also annotated the formal pattern, for convenience to the programmer.

In general, the BNF for unannotated parameter lists appear as a sequence of one or more argument types separated by semicolons, as shown below (with non-terminal symbols in italics):

$$
\begin{array}{rcl}
\textit{arg-list} & ::= & \textit{arg} \mid \textit{arg} \; ; \; \textit{arg-list} \\
\textit{arg} & ::= & \textit{primitive} \mid \textit{structured} \mid \textit{matrix} \mid \textit{pointer} \\
\textit{primitive} & ::= & \mathsf{BOOLEAN} \mid \mathsf{INTEGER} \mid \mathsf{FLOAT} \mid \mathsf{STRING} \ldots \\
\textit{structured} & ::= & \{ \; \textit{arg-list} \; \} \\
\textit{matrix} & ::= & \textit{arg} \; [\; \textit{index-list} \;] \\
\textit{index-list} & ::= & \textit{number} \mid \textit{number} \; , \; \textit{index-list} \\
\textit{pointer} & ::= & \uparrow \textit{arg}
\end{array}
$$

(Pointers to data are treated as just another set of data types, and in the next section we will discuss the impact pointers have on interface manipulations.) The annotated parameter list is denoted by a list of one or more *labeled* arguments separated by semicolons. The format is similar to that described above, except that each argument is uniquely labeled.

Argument components are also labeled, so that the programmer may access individual structure components. When programmers provide the annotated pattern, they can choose descriptive labels for any of the arguments or components, as long as the labels are unique. When programmers rely upon NIMBLE to annotate the interfaces, the labels will look much more mundane. To illustrate the convention, consider the parameter list described by:

a:INTEGER; b:{ a:INTEGER; b:BOOLEAN[10] }; c:{ a:INTEGER; b:STRING}[20]

This interface consists of three parameters: an integer, a structured argument (composed of an integer followed by an array of ten booleans), followed by an array of twenty structures (each composed of an integer followed by a string). One accesses the various elements by

a	- first argument in the actual parameter list.
b.a	- first component of the second actual parameter.
c[0].a	- first component of the first array element of the last actual parameter.
c[1].a	- first component of the second array element of the last actual parameter.
⋮	⋮
c[19].b	- last component of the last array element of the last actual parameter.

Throughout this paper, we will use our system-provided labels in all examples, unless otherwise noted.

A map created by the programmer is in terms of the labels of the actual pattern. Maps are very simple, a list of NIMBLE labels separated by semicolons, with optional brackets to indicate record structures. If the target interface pattern in Figure 1 was defined solely as the string-valued parameter Name, then, based on the annotated patterns given above, the NIMBLE map would be just that: Name. NIMBLE would then generate the appropriate adaptor so that just the right 'name' field would be plucked out of the caller's interface and transmitted to the intended procedure for processing. As it is, the target interface is slightly richer, so we would use the map in Figure 3, as will be described.

For the sake of consistency, the range of each map must match the structure in the formal pattern, and the NIMBLE translator will check that this is true before generating any adaptor. Two parameter lists are considered to be *structurally equivalent* if there exists a bijection between the lists such that (1) a primitive actual parameter only maps onto a formal parameter of the same primitive type, and (2) a composite actual parameter only maps onto a structurally equivalent formal parameter. If the bijection is order-preserving, such that the i^{th} actual argument maps onto the i^{th} formal argument, then the mapping is an *isomorphism*, and the parameter lists are said to be *syntactically equivalent*. We define the notion of *semantically equivalent* parameter lists to be lists that convey the same information, i.e., the same data values, though not necessarily in the same format or order. Henceforth, we use the general term *equivalence* to mean *syntactic equivalence*. (We will continue to fully specify *semantic equivalence*).

For the remainder of this section we review the various coercion scenarios that can be handled by the NIMBLE notation. Two parameter lists are semantically equivalent by the *commutative* property if there exists a bijection between parameter structures, but the parameters do not appear in the same sequence. Given the annotated actual interface pattern, one could use the NIMBLE notation to impose any reordering of arguments, thus creating a new parameter list that is semantically equivalent to the original actual parameter list, but is syntactically equivalent to the formal interface pattern:

Examples:

```
ACTUAL PATTERN:   a:INT; b:FLOAT
FORMAL PATTERN:   FLOAT; INT
NIMBLE:           b; a

ACTUAL PATTERN:   a:INT; b:{ a:FLOAT; b:STR; c:BOOL[10] }
FORMAL PATTERN:   { BOOL[10]; FLOAT; STR }; INT
NIMBLE:           { b.c; b.a; b.b }; a
```

Notice that in the second example, we need to reorder the components of the structured argument. Thus, the commutative property is recursive, affecting both arguments and their components.

Similarly, the existence or absence of structure does not alter the data values being passed. Two parameter lists are considered to be semantically equivalent by the *associative* property if, when we ignore all structure and simply consider the sequence of data objects, the resultant lists are equivalent. Thus, all three of the following interface patterns are considered to be associative.

```
{INT;INT};FLOAT      INT;{INT;FLOAT}      INT;INT;FLOAT
```

If the patterns are associative but not equivalent, the programmer must enforce the structural requirements imposed by the formal parameter pattern using the notation we have provided. Record structure is created by listing the components inside braces; matrix structure is formed by surrounding the array elements by square brackets; by placing a $\uparrow$ symbol before an argument, one creates a pointer to the argument. Structure can be relaxed by simply listing an actual argument's components without the associated structure.

Examples:

```
ACTUAL PATTERN:    a:INT; b:BOOL[3]
FORMAL PATTERN:    { INT; BOOL; BOOL; BOOL }
NIMBLE:            { a; b[0]; b[1]; b[2] }

ACTUAL PATTERN:    a:{ a:INT; b:STR }; b:INT[100]
FORMAL PATTERN:    { INT; STR; ↑INT[100]}
NIMBLE:            { a.a; a.b; ↑b }

ACTUAL PATTERN:    a:INT[3]; b:INT[3]; c:{ a:INT; b:INT }; d:INT
FORMAL PATTERN:    INT[3,3]
NIMBLE:            [ a; b; [ c.a; c.b; d ] ]
```

Not all actual and formal parameter lists are initially semantically equivalent. It may be the case that the invoking module transmits additional parameters that the invoked module will not accept. If the calling module transmits extraneous information, the programmer will need to *project* only the required data values out of the actual parameter list.

Examples:

```
ACTUAL PATTERN:    a:INT; b:STR
FORMAL PATTERN:    INT
NIMBLE:            a

ACTUAL PATTERN:    a:INT; b:{ a:STR; b:FLOAT[5,3]; c:STR }; c:BOOL
FORMAL PATTERN:    { STR; FLOAT[3] }; BOOL
NIMBLE:            { b.a, b.b[0] }; c
```

Of course, instead of manipulating existing actual arguments, the programmer may want to create new values for any of the passed arguments, especially when the calling module does not provide the required information. In this case, the programmer will need to *extend* the actual parameter list to include the missing data. The format for the creation of primitive data objects is the specification of the primitive type followed by the quoted value of the new object inside parentheses.

Example:

```
ACTUAL PATTERN:    a:INT
FORMAL PATTERN:    INT;BOOL
NIMBLE:            a; BOOL('TRUE')
```

Facilities are provided to allow the programmer to coerce any primitive data object into an alternate primitive type. We provide built-in functions for type conversions among all primitive types. The format resembles that of the creation of new data objects: the programmer specifies the primitive type followed by the actual data object inside parentheses.

Example:

```
ACTUAL PATTERN:   a:BOOL; b:FLOAT
FORMAL PATTERN:   INT; INT
NIMBLE:           INT(a); INT(b)
```

We have demonstrated how our NIMBLE notation can be used to permute an actual parameter list to match almost any formal interface pattern. However, we have avoided examples that require major alterations of array arguments. We can reference individual array elements and create small array structures, but most matrix transformations are complex and any attempt to describe them in a simple notation would require an enormous amount of patience. Consider the following example:

```
ACTUAL PATTERN:   a:INT[100]
FORMAL PATTERN:   INT[50]
```

The actual parameter is an array of one hundred integers, and the invoked module is expecting a vector of fifty integers. But which fifty integers? The first fifty? The last fifty? The components of the array whose index is even? We have provided a rule that allows the programmer to create an array by listing desired elements, but this is not feasible where large arrays are concerned.

We handle array manipulations with an EVAL statement, whose format is as follows:

EVAL (*id* , *parmlist*)

The EVAL rule states that the code in module *id* is to be executed with parameters *parm-list*, and the resultant data structure will be passed as a parameter. All EVAL modules used in a particular NIMBLE specification must be declared in an optional declarations section before the specification is given. These declarations include the name of the module to be called in EVAL; the number, order and type of the parameters the module is expecting; and the argument type of the result. NIMBLE checks that the parameter list in the EVAL statement is syntactically equivalent to the interface pattern provided in the declarations section. The resultant type is used when checking that the NIMBLE specification is syntactically equivalent to the formal interface pattern of the invoked module.

Examples:

```
ACTUAL PATTERN:    a:INT[100]; b:INT[100]
FORMAL PATTERN:    INT[200]
NIMBLE:            DECLS concat : INT[+];INT[+] − > INT[+]; END
                   EVAL(concat,a,b)

ACTUAL PATTERN:    a:INT[10,10]
FORMAL PATTERN:    INT[10]
NIMBLE:            DECLS diagonal : INT[+,+] − > INT[+]; END
                   EVAL(diagonal,a)

ACTUAL PATTERN:    a:INT[20,5]
FORMAL PATTERN:    INT[20]
NIMBLE:            DECLS
                       slice : INT[+,+];INT;INT;INT;INT − > INT[+];
                   END
                   EVAL(slice,a,0,19,0,0)
```

In the above examples, a module's function can be inferred by it's name. For example, the module concat accepts two arrays of integers and combines them into a single array. Procedure diagonal takes a square matrix and creates a vector consisting of the diagonal elements of the matrix argument. The slice module produces a submatrix, given a multi-dimensional matrix and new lower and upper bounds for each dimension of the original matrix. In our current implementation, the modules listed in the declaration sections and used in EVAL statements must be provided by the programmer. After some performance analysis, one could provide a library of commonly used coercion modules [7]. The reuse of these modules is as profitable as the reusability of the program modules whose interconnection is made possible by NIMBLE.

The EVAL statement is not intended to be a "catch-all" for situations that cannot be handled by the structure symbols and casting functions that comprise the NIMBLE notation. Our notation is complete, and can describe any combination of the algebraic permutations listed above. EVAL is provided because it is impractical to use the simple notation when dealing with large structures, such as matrices — it is feature of *convenience*. Along these lines, we also extended the above notation to allow arithmetic and boolean expressions wherever a primitive was allowed, with the exception of the declaration section of EVAL modules This allows *semantic* coercions of the arguments when needed. For example, if the actual interface pattern contains an array of 10 integers, and the called module is expecting a single integer, the expected integer may be the sum of the array elements rather than a particular array entry. This also provides the means to compute the index of the array element to be passed at run-time. Note that the use of expressions is not necessary for the solution of the parametric coercion problem (a syntactic problem); it simply provides a shorthand notation for simple computations that avoid the overhead associated with the use of EVAL.

If the module being invoked is a *function* (as opposed to a procedure), then NIMBLE is also responsible for coercing the function result, since NIMBLE controls all communication between the modules. We use the reserved word RETURN to separate the parameters from the resultant arguments.

Example:

```
ACTUAL PATTERN:    a:↑int; b:{a:str;b:bool[2][3]}[2] RETURN ↑{int;int }
FORMAL PATTERN:    ↑bool;str RETURN a:str
NIMBLE:            ↑b[1].b[0,2];str(a↑) RETURN ↑{int(foo);int('42')}
```

The ACTUAL PATTERN specifies the pattern of the parameters that the calling module is sending and the type of the result that it expects in return. The FORMAL PATTERN describes the parameters expected by the invoked module and the resultant type of the function. The NIMBLE specification describes the transformation from actual parameters to formal parameters, and the return transformation from the formal result to the actual result. Thus, we annotate the return portion of the FORMAL PATTERN. The above format must be used whenever the invoked module has a return value, since all data transfers between the two modules is managed by the NIMBLE module; this is true even if there is no need to coerce the resultant type.

A NIMBLE map to interconnect our example from Figure 1 is shown in Figure 3. This map combines several of the features just enumerated in this section. The Name and Address fields from the calling procedure are extracted from the record provided by the calling module. In addition, the Sex field coercion is performed by a programmer-provided function Usermap, which is applied by EVAL. This function must accept a single STRING valued parameter, and return an integer; the range of the function must match the semantics of the called function. Usermap can be written in any available application language. One example of such a map is given in Figure 4.

EXECUTION ENVIRONMENT

Once a map is constructed in NIMBLE, it may be used to generate an adaptor interconnecting two software components. With the NIMBLE declaration in a file, the user may now invoke the NIMBLE translator. But how NIMBLE proceeds from this point depends upon the target execution environment. Software developers integrate components within many different contexts (for example, components may be linked into a single process, or they may be distributed in a network). Information concerning the context of use affects how each adaptors is to be generated.

In order to support this diversity, we have implemented NIMBLE as a two phase translation process. Output from the translator's first phase is an operational specification of the execution-time steps necessary to transform the actual parameters into a structure that conforms with the formal interface pattern. This specification is given in a 'pseudo code' that is suitable for input to any of several possible implementations of the second NIMBLE phase. (Programmers ordinarily do not manipulate this intermediate form of the NIMBLE translation map.) The final result is an adaptor appropriate for the programmer's particular context of use, as described below.

CONTEXTS OF USE

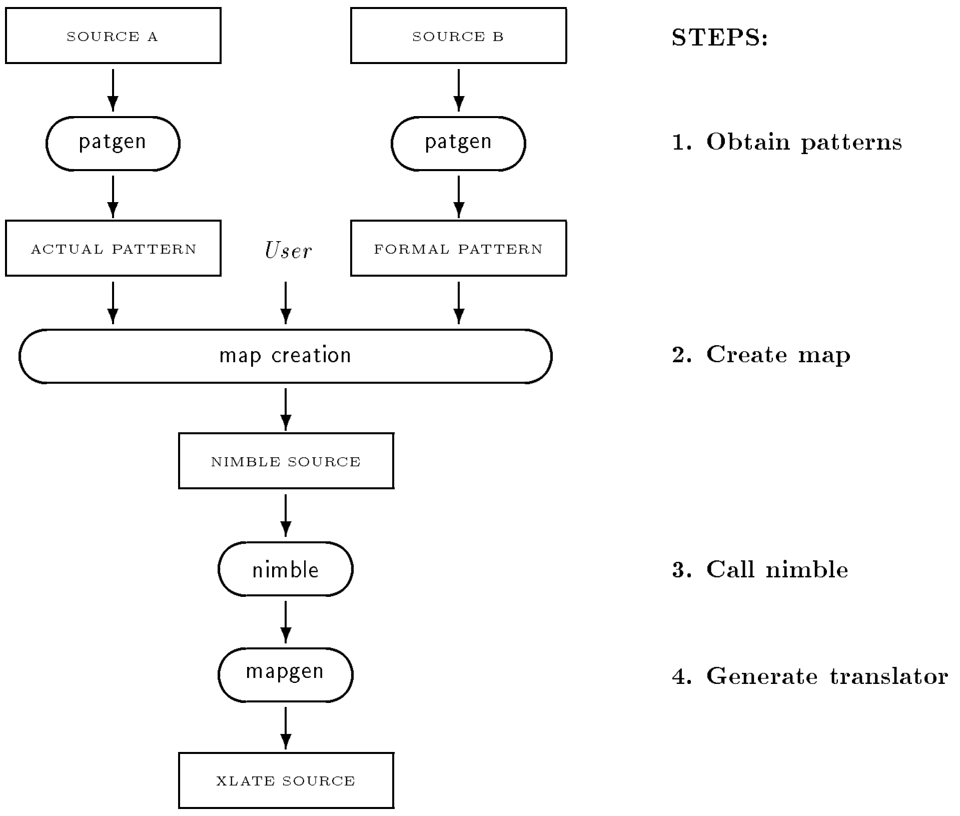
The most common use of NIMBLE is to assist integration of components into a single, same-language process. The adaptor is created statically, and linked into the application appropriately. For example, the code generated for our example problem would be translated into the C program:

```
struct Employee {                   xlate( a )
    char *Name;                     struct Employee a;
    char *Address[4];               {
    int Sex;                            int nimble_i;
    int Age;                            struct Addressee b;
    int SocNum;                         b.Sex = Usermap(a.Sex);
    float Salary; };                    b.Name = a.Name;
struct Addressee {                      for(nimble_i=0;nimble_i<4;nimble_i++){
    int Sex;                                b.Address[nimble_i] = a.Address[nimble_i];
    char *Name;                         }
    char *Address[4];                   xlate_inner( b );
};                                  }
```

Creating the source for an adaptor is a straight forward task, but its use involves some subtleties. Extensive aliasing which may occur: the procedure initiating the call will only know the name of the 'real' procedure to be called, not the intermediate NIMBLE procedure. The linkage editor is responsible for insuring that the actual calls performed will be first to the xlate routine above (or its equivalent, depending on the environment), and also for binding the xlate_inner call to the target procedure.

In general, designers have a great deal of flexibility in how the coercion routine is installed, but the use of NIMBLE in this environment relies upon the existence having a linkage editor that can handle aliased bindings. This requirement is satisfied (almost by definition) by most reasonable module interconnection languages and related facilities. The system we use for this capability is the Polylith software interconnection system [5]. The sequence of steps taken by a user to produce the above example is shown in Figure 5.

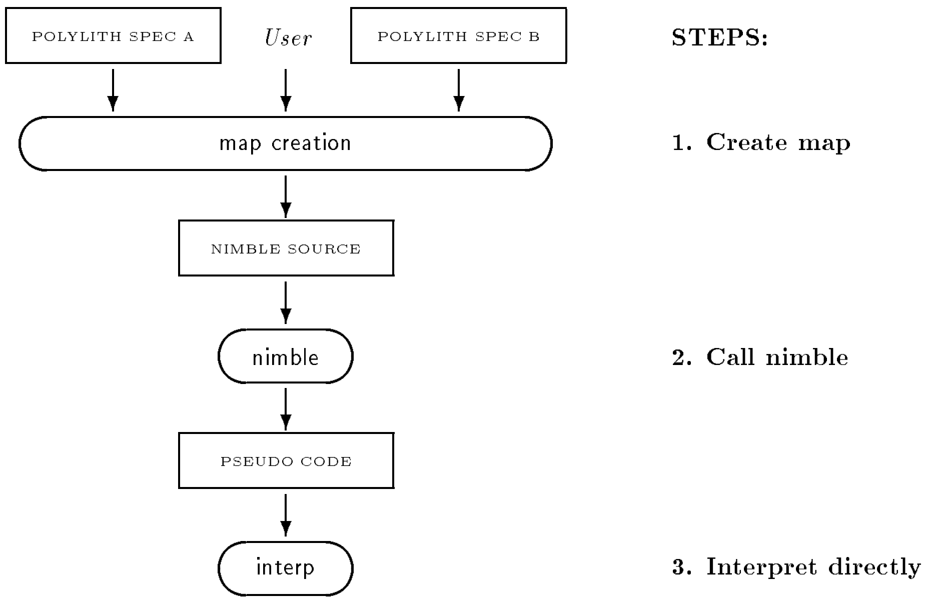
An alternative to the compiler-oriented, single process space execution environment described above is to interpret the pseudo-code directly. The NIMBLE map may be incorporated directly into the application design, as is typically expressed in the MIL in systems such as Polylith. At execution time, an interpreter is used to process the pseudo-code specification and perform the required parametric coercion. This



Description of steps:

1. Either user or support system invokes patgen, a filter created via newyacc, to extract the interface patterns for named procedures in a given application language. Alternately, the user can directly enter the pattern.
2. The user creates a desired map based on the two patterns. This step can be performed using normal text editors, or with a Suntools-based editor designed specifically to assist in this step.
3. The user invokes nimble to create the operational specification for this map. Equivalence checking between the formal pattern and the range of the map is performed here.
4. The user invokes one of a set of post-processors to generate a language-specific translation routine. Either the user or the support system must notify the interconnection system so that bindings of the call in A to B wil go through the translator.

Figure 5: Sample sequence of steps for preparing a translation with Nimble.



Description of steps:

1. A map is created as before. Fewer preprocessing steps are needed since the patterns are directly available in the Polylith structural specifications.
2. The user invokes nimble as before.
3. The operational specification is directly executable by an interpreter available for this task in the Polylith system. Only one such tool is needed since all representation questions are handled directly in Polylith's toolbus.

Figure 6: Sample interaction for using Nimble in a prototyping environment.

approach is especially effective in prototyping environments. Nimble can facilitate the direct (re)use of library modules with a minimal investment in 'pipe fitting.' The sequence of steps a user would take in order to obtain this effect in an interpretive environment is shown in Figure 6.

A unique use of the Nimble tool is one of data conversion of host files. The Nimble pseudo-code can be translated directly into a standalone tool which can be applied directly to data files. This is useful when a large file of data must be transformed in more elaborate ways than can be effected by tools such as, for instance in the Unix domain, *sed* [8] or *awk* [9]. Used in conjunction with a heterogeneous interconnection system such as Polylith, Nimble can be used to transfer and coerce data files across different system architectures.

TYPE CONSIDERATIONS

The question of how well we have chosen our set of primitive data types is an important one. Even for a 'simple' datum of type *integer* there are many, and often inconsistent, assumptions made about semantics. The host architecture's binary encoding scheme, the range of values possible, and behavior of operations on elements of the type are all potential sources of error, should our Nimble processors deal with the data differently than do the object codes for an application.

When all procedures are implemented in the same language and are linked into the same process space, the correspondence between Nimble's semantics for a type and the application language's semantics does not need to be strong. This is because the names of primitive types in Nimble end up acting simply as place holders for the declarations of that application language. The post-processor will simply generate a source code in the same language of interest. This also makes it easy to add new primitive types, as long as the implementor is able to correctly provide datum size information to the appropriate post-processor.

For mixed language programming, a system such as Polylith is necessary. In this situation, the Nimble type names again act as place holders, but for abstract data types rather than for predefined language types. The interconnection system is then responsible for providing representation functions at the point of the call to map the elements from the host and language specific representation to a 'standard' representation, and for providing inverse representation functions to map the formal arguments from the 'standard' representation to the specific representation expected by the procedure. Nimble merely coerces parameters within the 'standard' representation, without concern or knowledge of the specific data representation used by either the source or the destination module. This approach also makes distribution of the two procedures transparent; the call can be turned into an RPC with no change to the procedure implementation themselves. Techniques of this type are described in detail in Reference 5.

In each of the execution environments described above, pointers are handled 'as can be expected' depending upon the context. When all data passed is in the same process space, then the NIMBLE place holders for pointer data can be installed in a translated call consistently. When the call crosses process boundaries, then the pointer will only be correct if the underlying execution environment supports the additional address translation needed for non-local referencing. Few systems provide this capability. However, even without it, NIMBLE *will* allow correct dereferencing of pointers at the point of the call should the user's map direct that referenced data be accessed to build an immediate object for transmission to the non-local service (as long as the resultant coercion module lies in the same process space as the invoking module). The success of mapping from a data object to a referenced object *does* depend on the address translation capabilities of the underlying environment.

Finally there is the question of how well NIMBLE translation routines support the calling conventions of the application language domains involved. This is primarily dependent on the post-processor which generates source code based on our NIMBLE pseudo-code. In general, single address space environments can return values as one would expect when parameters are passed by reference. The exception to this occurs when simultaneously a parameter is passed by reference and the user directs that the value being referenced must in fact be accessed and coerced by NIMBLE before transmission to the callee. In this case, the post-processor can implement a 'copy back' scheme, where all changes to the parameter variable made by the procedure module are inversely coerced (to the actual's data representation) and placed in the caller's variable references. Thus, subsequent non-local references to the actual parameter made by the process executing the called procedure will accurately report the changed variable value.

Unfortunately, there are situations involving pointers where, when a user specifies that immediate data be created for transmission to the callee, the post-processor may allocate temporary storage to hold the intermediate values constructed by NIMBLE. Should a pointer to this storage be imprudently returned as the value of some other parameter in the pattern, then a 'core leak' can be established with disastrous consequences: the calling routine would have pointers to storage which the run-time environment might then reallocate for other uses. This type of error can really be classified as a heap management problem, and in many ways is a difficulty independent of any use of NIMBLE.

RELATED WORK

Related module interconnection language research has focused more closely on semantic correspondence of interfaces, e.g. the Inscape environment. However, there has been very little related work on the intermediate 'syntactic' level addressed by NIMBLE.

One data restructuring system called "CONVERT" was described in Reference 7. Expressed using our terminology, this was really an investigation into what functions one could want to have around to use via in an EVAL operation. It did not address transformations of interface patterns themselves. More recently, the ‘Q’ representation system in Arcadia does handle some aspects of interface adaptation between C and Ada [11], as does SLI [12].

Finally, some data transformation work has been performed in the database community. The PAL language (PRECI Algebraic Language) was developed to integrate information stored in different databases [13]. Although some of the data integration problems that PAL solves are similar to our coercion problem (type differences, missing data and structural differences), their solutions could not be adapted to the contexts of use which require substantial data transfer during invocation.

CONCLUSION

Nimble has been implemented in C and runs on most any 4.2BSD and 4.3BSD Unix systems. The output from Nimble is an adaptor written in C that can be linked into an application to perform the desired translations when called. To integrate components written in other languages, Nimble adaptors can employed via a mixed-language programming facility such as Polylith, although it would be simple to alter Nimble’s post-processor to generate source in other languages directly. Our experience with use of Nimble adaptors in tightly-coupled systems is that the performance costs are low — approximately the cost of performing one extra procedure call at each point where the adaptor is employed. This compares favorably with the extra development costs that would be needed to adapt those same interfaces manually.

The economic advantage of reusing software rather than reimplementing it is highly sought after. Most investigations in this line focus on ways to adapt software (or design specifications) to meet a new context of use. Nimble demonstrates that it is easy and valuable to also be able to adapt the context of use to meet the needs of our existing modules.

REFERENCES

[1] V. Basili. Software development: a paradigm for the future. *Proceedings of COMPSAC '89*, (September 1989), pp. 471-485.

[2] M. Lubars and M. Harandi. Intelligent support for software specification and design. **IEEE Expert**, vol. 1, (1986), pp. 33-41.

[3] P. Ward. The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. **IEEE Transactions on Software Engineering**, vol. 12, (February 1986), pp. 198-210.

[4] B. Joo. Adaptation and composition of program components. Doctoral Dissertation, University of Maryland, (1990).

[5] J. Purtilo. The Polylith software bus. University of Maryland Technical Report, TR-2469 (1990).

[6] J. Purtilo and J. Callahan. Parse tree annotations. **Communications of the ACM**, vol. 32, (1989), pp. 1467-1477.

[7] N. Shu, B. Housel and V. Lum. CONVERT: A high level translation definition language for data conversion. **Communications of the ACM**, vol. 18, (October 1975), pp. 557-567.

[8] L. McMahon. SED- A non-interactive text editor. *AT&T Bell Laboratories*, 1978.

[9] A. Aho, B. Kernigan and P. Weinberger. Awk- A pattern scanning and processing language. *AT&T Bell Laboratories*, (second edition) 1978.

[10] D. Perry. The Inscape environment. *Proceedings of 11th Intl Conf on Software Engineering*, (May 1989), pp. 2-12.

[11] M. Maybee. Q: A multi-lingual interprocess communications system for software environment implementation. University of Colorado Technical Report, CU-CS-476-90, (1990).

[12] Wileden, J., A. Wolf, W. Rosenblatt and P. Tarr. Specification level interoperability. *Proceedings of 12th Intl Conf on Software Engineering*, (March 1990), pp. 74-85.

[13] S. Deen, R. Amin and M. Taylor. Data Integration in Distributed Databases. **IEEE Transactions on Software Engineering**, vol. 13, (July 1987), pp. 860-864.

ACKNOWLEDGEMENTS

We appreciate the suggestions given to us from Victor Basili, Ralph Jeffords, Sean O'Malley and Marvin Zelkowitz. This research was concluded with support from Office of Naval Research under contract number is N0014-90-J4091.