Avoiding Packaging Mismatch with Flexible Packaging

Robert DeLine, Member, IEEE

Abstract—To integrate a software component into a system, it must interact properly with the system's other components. Unfortunately, the decisions about how a component is to interact with other components are typically committed long before the moment of integration and are difficult to change. This paper introduces the Flexible Packaging method, which allows a component developer to defer some decisions about component interaction until system integration time. The method divides the component's source into two pieces: the ware, which encapsulates the component's functionality; and the packager, which encapsulates the details of interaction. Both the ware and the packager are independently reusable. A ware, as a reusable part, allows a given piece of functionality to be employed in systems in different architectural styles. A packager, as a reusable part, encapsulates conformance to a component standard, like an ActiveX control or an ODBC database accessor. Because the packager's source code is often formulaic, a tool is provided to generate the packager's source from a high-level description of the intended interaction, a description written in the architectural description language UniCon. The method and tools are evaluated with a series of experiments in which three wares and nine types of packaging are combined to form thirteen components.

Index Terms—Mismatch, software packaging, system integration, software component, software architecture.

1 INTRODUCTION

In order to reuse a software component, not only must a developer consider what the component computes, but also how it makes that computation available to other components. A component that expects to interact with other components through procedure calls, for example, is difficult to reuse in a system where components interact by exchanging messages, by raising and listening for events, or by accessing data in shared memory. The assumptions a component makes about how it interacts with other components constitutes its packaging. Today, reusing a component in a new system requires attention to both its functionality and its packaging.

The motivation for selecting a component for reuse is typically its functionality. Often the only preference a developer has about packaging is that it be appropriate to the system in which the component is to be integrated. A Windows developer, for example, will shop for an ActiveX control with the desired functionality for his application, whereas a Unix developer will look for a filter. Today's off-the-shelf components come prepackaged: Decisions about the component's packaging are made at development time, before the component is made available for reuse. When the packaging decisions encapsulated in a reused component are unsuitable in the context of a new system, the condition is called packaging mismatch [13], [27].

When packaging mismatch occurs, the system integrator must undo or circumvent the unsuitable packaging decisions, which is often an expensive proposition. In the source code of a conventional software component, the code that accomplishes the interactions tends to be interspersed with the code that accomplishes the component's functionality. This makes it difficult to identify the code related to packaging. When changing the source code is infeasible or overly expensive, the system integrator typically overcomes packaging mismatch by introducing "glue code" in the form of wrappers or mediators [11], [21]. For example, if a component packaged to interact through procedure calls is to be used in an event-based system, the system integrator might place a wrapper around the component that receives events and that makes the appropriate procedure calls on the wrapped component. This glue code becomes another part of the system to test and maintain.

The heart of the packaging mismatch problem is that engineering decisions are being made too early, when too little of the relevant information is known—a violation of Parnas' widely accepted information hiding principle [22]. Since packaging decisions are largely about system integration, they should be deferred until the information about the integration context is known. This paper introduces a method, called Flexible Packaging, for structuring a software component's source code to defer decisions about interaction until integration time. Of course, not all decisions about interaction can be deferred: A component's functionality cannot be expressed without mentioning some aspects of interaction. Flexible Packaging provides a mechanism for specifying those aspects of interaction that are essential to the functionality, while deferring the incidental details.

As one would expect, every component cannot be packaged in every way; the fact that the functional and interactive concerns can be separated, does not imply that arbitrary mix-and-match between them is feasible [12]. This paper focuses on the mechanism that achieves the separation between these concerns. Describing abstract patterns of
interaction based on this mechanism, which would allow compatibility checks between the functionality and packa-
compatibility between the functionality and packaging, is the next phase of the research.
The remainder of this paper discusses Flexible Packaging in more detail. Section 2 uses a simple example to contrast the current practice of packaging components with Flexible Packaging. Section 3 explains the technology and tools behind Flexible Packaging. Section 4 discusses our use of case studies to evaluate the method. Section 5 reviews related research. Section 6 discusses future work and Section 7 concludes.

2 Flexible Packaging in Action
To illustrate the problems with today’s component development and integration and to contrast current practice with the Flexible Packaging method, this paper uses an example component that is small enough to show complete source code, but representative of a larger class of computations. The example component’s function is to compute the arithmetic mean of a set of scores for each student enrolled in a course. This could be expressed in pseudocode, as in Fig. 1a. We will consider two packagings for this component. The first version uses COM to read the scores from a Microsoft Excel spreadsheet and to write the means to the spreadsheet. The second is a Unix-style filter which reads student records, as formatted in Fig. 1c, from standard input and reports the means to standard output.

2.1 Current Practice: Mixing Concerns
The source code for the spreadsheet version of the averaging component is shown in Fig. 1b; the source code
for the filter version, a script for the parser generator Yacc, is shown in Fig. 1d. Several important aspects of today’s practice can be readily seen in this example. First, in both versions, the code that implements the packaging and the code that implements the functionality are completely intermingled. Indeed, even visually spotting the key lines from the pseudocode version is difficult. (The change bars at the right of the figure highlight those lines of code.) Second, the implementation of the packaging can cause the expression of the component’s functionality to be obscured. For example, the arrangement of the code that Yacc induces in Fig. 1d has little relation to the pseudocode in Fig. 1a.

Third, the two versions of the code are very different from one another. Changing the implementation from one version to the other would require effort.

### 2.2 Separating the Concerns

In contrast to current practice, the Flexible Packaging method advocates and supports the separation of a component’s functionality and its packaging into distinct software artifacts. The component’s functionality is encapsulated in a reusable part called a **ware**; its packaging, in a reusable part called a **packager**. The ware and packager, when compiled together, form the complete component.

Fig. 2a shows the source code for the ware for our averaging component. It is written in an extension to the C programming language, called Ciao, which has high-level constructs for describing the intended interaction with other components. Packers are also written in Ciao. However, rather than being implemented by hand, a packager’s source code is automatically generated from a high-level specification of the packaging, called a **packaging description**, written in the architectural description language UniCon [28].

The packaging description in Fig. 2b specifies a component packaging that accesses a spreadsheet. It reads a series of cells (“Names”) from row 2 to the first empty row of column A. For each of these cells, it first reads a series of cells ("Scores") from column B to the first empty column of the current row and then writes a cell ("Mean") in the first empty column of the current row. Fig. 2c specifies a component packaging that acts as a filter with one input stream and one output stream. The input stream contains data formatted according to the extended regular expression

\[
\text{name} : [a-zA-Z]+ \\
\text{id} : [0-9]+ \text{ (score [0-9]+ : [0-9.]+)*}.
\]

The output stream consists of lines, each of which contain a name and mean, separated by a space.

To create the spreadsheet version of the averaging component, the system integrator runs the Flexible Packaging tools on the ware in Fig. 2a and the packaging description in Fig. 2b. From these two, the tools produce a component whose behavior is the same as the handmade component in Fig. 1b. Alternatively, she could give the same ware and the packaging description in Fig. 2c to the tools in order to produce the filter version of the component, which behaves like the filter in Fig. 1d. In both cases, the same ware is reused without modification. Note also that the ware’s source code resembles the pseudocode from Fig. 1a and is not fragmented as it is with the handmade components.
With the Flexible Packaging method, vendors no longer sell reusable pieces of functionality in the form of prepackaged components; instead, they sell wares. In turn, a system integrator acquires wares with the needed functionality and selects an architectural style for her system (perhaps for the systemic properties that the architectural style brings, perhaps for compatibility with legacy components). She then creates a packaging description for each ware to make the ware suited to that architectural style. For instance, if the integrator were using the averaging ware in a system with Excel spreadsheets, she would write the spreadsheet packaging description to create the spreadsheet version of the component; if she were using the ware in a pipe-filter system, she could write the filter packaging description to create the filter version. In this way, tailoring a component’s packaging to the integration context avoids packaging mismatch.

3 THE FLEXIBLE PACKAGING METHOD AND TOOLS

The Flexible Packaging method takes the decisions about a component’s packaging out of the hands of the component provider and puts them into the hands of the system integrator. There are two key elements in the design of the Flexible Packaging method that accomplish this shift in responsibility. First, to prevent the system integrator from being burdened with the uninteresting details of component packaging, we introduce a new participant in the method: the packaging specialist. Second, to allow a ware to be combinable with many different packagers (and vice versa), the ware and packager export an interface based on coroutines and shared data channels rather than procedure calls.

3.1 Encapsulating Packaging Details

Today, in order for a component provider to ensure that his component has a particular packaging, he typically exercises specialized skills and knowledge to determine both the content of the component’s source code and the software construction steps used to process that source code. Consider the list of requirements a component must meet to be packaged as a Netscape plug-in: It must implement sixteen particular functions and use memory management functions that Netscape provides; it must be compiled into a dynamically linked library (DLL) that exports three particular functions and that contains a resource fork with two particular text resources; the DLL’s name must be in DOS 8.3 format and begin with the letters NP; finally, the DLL must appear in a particular directory. Of these requirements, the first is about the content of the component’s source code; the rest, about its construction. Further, the byzantine nature of these requirements is typical of many packagings.

The knowledge and skills needed to achieve a component packaging are different from those needed to achieve a given piece of functionality. For instance, if a component provider creates a Netscape plug-in that animates objects based on differential equations, the knowledge needed to create the functionality (the convergence properties of various numerical methods, the effects of floating-point round-off, and so on) is distinct from knowing the plug-in requirements above. Shifting packaging decisions away from the component provider keeps him from being burdened with skills and knowledge that are not relevant to his job.

However, these packaging details are also not relevant to the system integrator, whose job is to understand the system-level implications of choosing different architectural styles. Exposing the integrator to this packaging complexity would be especially burdensome to those integrators who build their systems in heterogeneous architectural styles, which at once involve several types of packaging. Hence, in shifting packaging decisions from the component provider to the system integrator, we do not also want to shift responsibility for the code and construction implications of those packaging decisions.

To avoid burdening the system integrator with these packaging details, we introduce a new participant in the method: the packaging specialist. A packaging specialist is an expert in a given packaging technology, like Netscape plug-ins, and encapsulates the arcane rites necessary to ensure that a component has that packaging. How should this encapsulation be achieved? To motivate Flexible Packaging’s answer, three characteristics of the packaging specialist’s job are noteworthy. First, as previously mentioned, the packaging specialist is responsible not only for the source code needed to achieve a given packaging, but also for the necessary noncode artifacts (like a plug-in’s resource fork) and the construction and installation steps. Second, the products for which the packaging specialist is responsible are not fixed, but vary according to the packaging decisions that the system integrator makes. For instance, with Netscape plug-ins, the system integrator chooses the MIME type and file extension that the plug-in handles. The packaging specialist is then responsible for producing the source code and other products that reflect the system integrator’s choices. The packaging specialist encapsulates the syntax and tools associated with these products, but must allow the content to vary with the integrator’s decisions. Third, because the system integrator makes her high-level packaging decisions to tailor the component to the integration context, she makes these decisions at system integration time. Hence, the products for which the packaging specialist is responsible cannot be produced until system integration time.

Given these three characteristics of the packaging specialist’s job, what should the Flexible Packaging method require the packaging specialist to produce to encapsulate the low-level packaging details? A fixed module or module library would be inadequate because it fails to capture the variability in the source code that results from the integrator’s high-level packaging decisions. Modules with parameters (like SML functors or Ada generics) are capable of capturing the variability in this source code, but are not applicable to the noncode artifacts and the construction and installation steps, since these are not written in programming notations.

Instead, the Flexible Packaging method requires the packaging specialist to produce a software generator. This generator maps a set of packaging parameters to the set of source code, noncode artifacts, and construction and
installation steps that are needed to achieve the packaging. The source code that the generator produces is called the packager. The parameters that capture the system integrator’s packaging decisions are written in the form of a packaging description, examples of which were shown in Fig. 2b and Fig. 2c. This packaging description is really a compound parameter, like a record in a standard programming language. The packaging generator reads this parameter and from it generates the necessary products and construction steps. Because a packaging description is just a compound parameter to a generator, any self-describing data notation, like XML [31], could be used to encode the information in the packaging description. However, UniCon is a particularly suitable choice since it is a self-describing data notation that is geared toward architectural abstractions, like component interfaces (packagings). Hence, by encoding packaging descriptions in UniCon, they can be used not only with packaging generators, but also with UniCon’s tools, like its box-and-line architecture editor.

The packaging specialist’s generator takes only the packaging description—and in particular not any ware—as its input. In part, this reflects the facts that a given component standard is compatible with many kinds of functionality. The requirements for a component to be a Netscape plug-in, for example, are the same, whether the plug-in paints an image, displays a document, or draws nothing at all. Because the packaging generator is not relative to any ware, it can be used with many different wares, which amortizes the expense of developing it. Determining whether a generated packager is compatible with a given ware is the subject of the next two sections.

In summary, achieving a particular component packaging involves both making packaging decisions and deriving detailed artifacts from these decisions. The Flexible Packaging method places the responsibility for the packaging decisions in the hands of the system integrator. To keep the integrator from being overburdened, the method introduces the role of the packaging specialist, who provides a generator to map the integrator’s packaging decisions to the detailed artifacts.

3.2 Mixing Wares and Packagers

The packager and the ware each have a computation associated with it. The packager’s computation achieves the interaction (for example, access to the spreadsheet); the ware’s computation achieves the functionality (the calculation of the mean scores). Given that these two computations must be combined to achieve the component’s total behavior, what mechanism should be used to coordinate the two computations, i.e., to allow them to exchange control and data? The key to Flexible Packaging’s answer is the observation that ware providers and packaging specialists work independently of one another. A ware provider cannot anticipate all of the packagings with which his ware will be combined, nor can a packaging specialist anticipate all of the wares with which his (generated) packagings will be combined. Given that the packager and the ware are to be produced independently and that the ware should be usable with a variety of packagings (and vice versa), our chosen mechanism must support this independence.

The most ready choice is the mechanism that today’s tools support best and that is best understood in practice, namely the procedure call (or its relatives, method call and higher-order function call). Using this mechanism, the packager and the ware could each be encapsulated in its own module, with procedure calls between them for exchanging control and data. Unfortunately, this choice of mechanism violates our desire to reuse the ware with multiple packagings and vice versa, as the following example illustrates.

3.2.1 A Procedural Interface between the Packager and Ware

Fig. 3 shows the two versions of the averaging component, each implemented as a ware and a packager, with a procedural interface between them. The ware Ws and the packager Ps were designed to work together and implement the same functionality as the handmade component in Fig. 1b; similarly, the ware Wf and the packager Pf work together to implement the same functionality as the handmade component in Fig. 1d.

Each component’s procedural interface was designed independently; hence, the two interfaces are quite different. With the spreadsheet component, the ware drives the computation, pulling the scores from the packager and pushing the mean to the packager. With the filter component, the packager (the Yacc-generated parser) drives the computation, pushing the scores to the ware and pulling the mean from the ware. (Without the use of Yacc, the filter component’s source could be made similar to the spreadsheet component’s. However, as is, the Yacc version is representative of the “external control” structure that message passing, events, and user interfaces typically induce.) The direction of the procedure calls reflects the data push/pull distinction.

Once a ware has been written against a particular packager’s procedural interface, combining it with a different packager requires either that the ware be rewritten or that “glue code” be inserted between the ware and the packager to overcome the differences between the old interface and the new. As Black points out, the glue code between two active computations, like Ws and Ps, must act as a buffer; the glue code between two passive computations, like Wf and Pf, must act as a pump [3]. In both cases, the glue code is not negligible and performs a significant computation. In contrast, the Flexible Packaging method promotes a modularity mechanism that greatly reduces the need for such glue code.

3.2.2 A Channel-Based Interface between the Packager and Ware

To promote the desired mix-and-match between packagings and wares, the Flexible Packaging method provides a modularity mechanism based on data channels and coroutines. Packagings and wares are written in an extension to the C programming language, called Ciao, with a few additional constructs:

- `channel [in | out] [stream] <type> <cname>;`
- `in(<cname>, <varname>);`
- `out(<cname>, <expression>);`
- `alt { (<in> : <statements>) | }`
The first of these constructs declares a channel, which is a communication medium between the packager and ware. If one of the two declares the channel as `in` and the other as `out`, then data flows unidirectionally across the channel from the `out` declarer to the `in` declarer. The `in` statement is used for receiving data from a channel; the `out` statement, for sending data to a channel. Both the dataflow direction and the type of the data specified in an `in` or `out` statement must be consistent with the channel declaration. It is erroneous, for example, for an `out` statement to name a channel declared as `in` and it is illegal for an `out` statement to send a floating point value on an `out` channel of type int.

Stream channels are used to communicate multiple values between the ware and packager. Stream channels support an operation to express that no more values will be written to the channel ("close") and another to test for this condition ("more"). The `alt` construct, like its namesake in occam [17], allows an input to occur from any one of a set of `in` statements for which input is ready.

The ware and packager computations are interleaved through coroutines. The component as a whole has a single thread of control that is passed back and forth.

---

Fig. 3. The two components from Fig. 1, each reimplemented as a ware and a packager that interact through a procedural interface. Combining `Ws` and `Ps` or `Wf` and `Pf` is a matter of linking, whereas combining `Ws` and `Pf` or `Wf` and `Ps` requires either rewriting the code or creating "glue code."
between the ware's computation and the packager's. The thread is switched between computations whenever the currently running computation either performs an `out` statement or performs an `in` statement on an empty channel. This allows one of the computations to produce a value that the other may immediately consume. The thread of control always begins in the packager's computation since the packager governs the component's interactions and exchanging a thread of control among components is a form of interaction.

Fig. 4 shows the two versions of the averaging component, implemented as a ware and two packagers, with a channel interface between the ware and packager. (The ware code is the same as in Fig. 2a; the packager code for `Ps` and `Pf` are automatically generated from the packaging descriptions in Fig. 2b and Fig. 2c, respectively.)

```
(Ware)
channel in stream char* Names;
channel in stream float Scores;
channel out stream (char*, float) Means;
char* name;
float score;

while (more(StudentNames)) {
  int count = 0;
in(StudentNames, name);
  while (more(Scores)) {
    in(Scores, score);
    total += score;
count++;
  }
mean = total/count;
out(Means, name, mean);
}

(Ps)
#define cellref(C,R) ( sprintf(cell, "%c%d", C, R), cell )

channel out stream char* Names;
channel out stream float Scores;
channel in stream (char*, float) Means;

void main(int argc, char** argv) {
  _ApplicationPtr app;
  _WorksheetPtr sheet;
  char cell[10], col, *name; float mean; int row;

  app.CreateInstance(L"Excel.Application.8");
  app->Visible = VARIANT_TRUE;
  sheet = app->Workbooks->Open(argv[1])->ActiveSheet;
  for (row=2; sheet->Range[cellref('A', row)]->Value.vt != 0; row++) {
    total = 0; count = 0;
    _bstr_t_bstrname = (_bstr_t)sheet->Range[cellref('A', row)]->Value;
    out(Means, (char*)bstrname);
    open(Scores);
    for (col='B'; sheet->Range[cellref(col, row)]->Value.vt != 0; col++) {
      _bstr_t_value = (_bstr_t)sheet->Range[cell]->Value;
      out(Scores, atof((char*)value));
    }
  close(Scores);
  in(Means, name, mean);
sheet->Range[cellref(col, row)]->Value = mean;
}
  app->Quit();
}

(Pf)
%{
channel out stream char* Names;
channel out stream float Scores;
channel in stream (char*, float) Means;
float m; char* n;
}%

%token STRING
%token NL /* newline */
%token INT
%token FLOT

%{

start: student start | student ;
student: name id scores { in(Means, n, m); printf("%s %fn", n, m); } ;
nname: "name ": STRING NL { out(Means, $3); open(Scores); } ;
id: "id": INT NL ;
\scores: score scores | score { close(Scores); } ;
score: "score" INT ":" FLOT NL { out(Scores, $4); } ;

main() { yyparse(); }
}
```

Fig. 4. The two components from Fig. 1, each reimplemented as a ware and a packager that interact through a channel interface. The ware, which is the same code as Fig. 2a, may be combined with either packager.
DELINE: AVOIDING PACKAGING MISMATCH WITH FLEXIBLE PACKAGING

Note that neither the ware source code nor the packager source code is fragmented between procedures, which is not the case with a procedural interface.

The Flexible Packaging toolset includes a Ciao compiler, which translates Ciao into C. The channels are implemented with dynamic arrays; the coroutining, with the Windows NT Fiber library (lightweight threads). Ciao was implemented as a language extension rather than a C library to allow channels to be type-checked.

3.2.3 The Use of Additional Coroutines

For simplicity, the previous section describes the channel mechanism as though there were always two coroutines, one for the ware and one for the packager. In fact, Ciao allows an arbitrary number of coroutines to be declared. The thread is scheduled among the unblocked coroutines in round-robin fashion. The ability to have more than two coroutines proved to be handy in three ways.

First, although a ware or packager can use an \texttt{alt} statement to express that it is willing to handle input on multiple channels, sometimes it is always willing to handle input on a given channel. For example, a packager may be willing at any time to accept input on an error channel and to report it in some packaging-specific way (e.g., by adding an entry to a log file or by popping up a dialog box). Given the mechanism described so far, each \texttt{in} statement in such a packager would have to be written as an \texttt{alt} statement that accepts input on both on the original \texttt{in} channel and the error channel. Instead of this tedious approach, the module can instead create a dedicated coroutine that loops reading the error channel. This allows the packager’s error-handling code to be written separately from its “normal path” code.

Second, multiple coroutines are used to support heterogeneous packagings. In practice, many components participate simultaneously in more than one style of interaction. For example, a component may simultaneously be an ActiveX control, file accessor, and database accessor. Such a component is said to have a heterogeneous packaging. To account for this heterogeneity, I split component packagings into two categories: \textit{Component Standard} packagings govern how a component’s services are invoked; \textit{Data Access} packagings govern how a component accesses external data in order to implement its services. Then, a component’s complete packaging consists of \textit{exactly one} Component Standard packaging plus \textit{zero or more} Data Access packagings. This packaging composition rule allows a single component simultaneously to be an ActiveX control, file accessor, and database accessor, but prevents a single component from simultaneously being an ActiveX control and a Netscape plug-in (which is not a sensible combination).\footnote{Flexible Packaging does allow an integrator to package a ware as both an ActiveX control and a Netscape plug-in to yield \textit{two separate} components.}

To support heterogeneity, the packaging generator for each Data Access packaging follows a convention. To implement the data access, it generates a coroutine whose first statement is an \texttt{in} statement on a “guard” channel, after which appears the code that starts the data access (e.g., that opens a socket or database). For example,

```c
void main(int argc, char* argv[]) {
    coroutine ReadFile() {
        int dummy; FILE* fp;
        channel in StartRead;
        in(StartReadFile, dummy); /* guard */
        fp = fopen(¨file-to-be-read¨, ¨r¨)); /¨...etc...*/
    }
}
```

The Component Standard packaging generators follow a complementary convention. At the point in the generated code at which the data access should occur, the generator produces an \texttt{out} statement on the appropriate guard channel. For instance, if a batch program packaging contains a file-reading interaction, the generated code would look like this:

```c
void main(int argc, char* argv[]) {
    out(ProgramName, argv[0]);
    /* normal batch interactions */
    ... out(StartReadFile, 1); /* trigger guard */
    ...}
```

This convention of using multiple coroutines and guard channels means that multiple packaging generators can combine their output by exchanging the names of the guard channels rather than by knitting together the generated code.

The third use of additional coroutines is discussed in the next section.

3.2.4 Additional Channel Features to Support Mix-and-Match

With the channel features described so far, there is still the opportunity for a mismatch between the packager and the ware. Given that the packager and the ware are independently written in Ciao—typically by two different developers who do not collaborate—there is no reason to assume that their use of channels will be consistent. In particular, the ware and packager may be inconsistent from one another in four aspects of their channel use:

- They may differ in the order in which they use the channels (ordering mismatch). For example, the ware may do an \texttt{in} on channel A then an \texttt{in} on channel B; whereas the packager may do an \texttt{out} on channel B then an \texttt{out} on channel A.
- They may use different names for the same channel (name mismatch). For example, the ware could call a channel “Init,” whereas the packager calls it “Begin.”
- They may differently represent the data on a channel (datatype mismatch). For example, the ware may send an ASCII string on a channel; whereas the packager expects a Unicode string.
- They may use different numbers of channels to interact (aggregation mismatch). For example, the ware may send one integer apiece on two channels, whereas the packager expects to receive a pair of integers on one channel.
Following the design maxim of making the common case simple, Flexible Packaging provides a cascade of techniques to allow the system integrator to overcome mismatches between the ware and packager.

In order to accommodate ordering mismatch, Ciao channels are asynchronous, rather than the synchronous channels of occam. If Ciao channels were synchronous, the ware and packager would have to agree exactly on the order in which they use channels. Instead, `out` statements buffer their data until the corresponding `in` statement occurs; an `out` statement never blocks. (Clearly, `in` statements still block until their corresponding `out` statements happen; otherwise, there would be no value to assign to the variable in the `in` statement. If both the ware and packager are blocked on `in` statements, the resulting deadlock is detected at runtime and reported.)

Although this semantics does accommodate a certain amount of ordering mismatch, there is a price for this looseness: A computation committing an `out` statement has no guarantee about when, if ever, its sister computation will “react” to the output it has given. Indeed, with asynchronous channels, the Ciao compiler does not even insist that there be an `in` statement corresponding to every `out` statement. There is also the possibility of deadlock: Both the ware and the packager could be blocked on an `in` statement, each needing data from the other in order to make progress. The next section discusses how this deadlock is detected.

Name and datatype mismatches are anticipated to be quite common. Notice that, since the names of the packager’s channels are derived from names appearing in the packaging description, the integrator can often avoid name mismatch altogether by selecting names for the packaging description that are compatible with the ware’s channel names. To handle the mismatch that does arise, the Ciao compiler accepts an explicit map between ware and packager channel names. For each pair of names in this channel map, the map may optionally contain a small Ciao program to overcome datatype mismatch. For example, if the ware contains this channel declaration

channel in double Grade;

and the packager contains this channel declaration

channel out char* Score;

a map entry that unifies these two channels would look like this

(Grade, Score, TypeFixupCode("  
channel in char* String;  
channel out double Real;  
char* s;  
double r;  
in(String, s);  
r = atof(s);  
out(Real, r); ");)

The fix-up Ciao code must contain exactly one `in` and one `out` channel declaration. Then, the Ciao compiler unifies the fix-up code’s `out` channel with the mismatched code’s `in` channel and vice versa (e.g., Score and String are unified and Grade and Real are unified). Having the fix-up code contain its own channel declarations allows the possibility of building up libraries of fix-up code for use in multiple channel maps. Given a map entry with fix-up code, the Ciao compiler inlines the fix-up code wherever an `out` statement appears in the original mismatched code (e.g., all `out` statements on the channel Score). The use of a declarative map to overcome name and datatype mismatch is similar to that used in Nimble [23] and to Yellin and Strom’s adaptors [32], which also handle ordering mismatch. Yellin and Strom, working in the context of languages with synchronous method invocation, use explicit buffering to overcome ordering mismatch rather than having an asynchronous communication model.

Finally, when combining the packager and the ware, the system integrator can add coroutines to overcome mismatches that are too complex to be handled with channel maps, such as aggregation mismatches. For instance, to merge two channels in the ware to a single channel in the packager, the integrator would write a coroutine that does `in` statements on the two ware channels and `out` statements on the packager channel, multiplexing the data in whatever way she finds appropriate. Similarly, if a ware has an `in` channel for which there is not corresponding `out` channel in the packager, the integrator could write a coroutine that introduces a new `out` channel on which a default value is written.

In short, Flexible Packaging provides a variety of techniques for overcoming mismatches between the ware’s and packager’s channels. The effort needed to use the technique is in proportion to the anticipated rareness of the mismatch.

3.3 Ware/Packager Compatibility

As mentioned earlier, arbitrary mix-and-match between wares and packagers is not possible. Some combinations of functionality and packaging simply do not make sense. For instance, packaging an animation program as a Netscape plug-in is reasonable, whereas packaging it as a Unix filter is not. The animation program requires the ability to paint to the screen, which a Netscape plug-in can provide and a Unix filter cannot. Determining whether a ware and a packager are compatible relies on a construct called a channel signature.

3.3.1 Channel Signatures

As described earlier, a ware or a packager is a kind of module that interacts with other modules through Ciao channels. (The term Ciao module will be used to refer neutrally to either a ware or a packager.) A channel signature is an interface description for a Ciao module, much like a signature in SML, an interface in Modula 3, or an IDL file in COM. A channel signature consists of two parts, a signature process and a channel list.

A signature process gives all possible orders in which the module might perform `in` and `out` statements at run time and is specified in the following subset of Hoare’s CSP [16], where `id` is an identifier in the style of C:
3.3.2 Checking Ware/Packager Compatibility

Given the task of determining whether a given ware and packager are compatible, the system integrator can proceed in three stages. First, she can use her understanding of the functionality and packaging to be combined and be guided by the ware’s and packager’s channel signatures. For an obvious mismatch, like the animation program to be packaged as a Unix filter, she may recognize from experience that this combination does not make sense. A quick glance at the channel signatures’ use of the internal and external choice operators may also reveal an incompatibility.

(Formalizing this first intuitive step to determine compatibility might be possible given a level of abstraction above that of the channel signature. For example, we could consider the space of all possible Ciao modules as having centroids or pure forms. We could then classify a Ciao module by saying which pure form it is most like or by saying that it is a hybrid of two or more pure forms. A set of pure forms might include the following:

- An abstract machine accepts commands from the outside world and produces results based on the command it is given. It has a channel signature of the following pattern:
  \[ AM = \text{in}(\_\_) \rightarrow \text{out}(\_\_) \rightarrow \text{AM} \]
  \[ \[] \text{in}(\_\_) \rightarrow \text{out}(\_\_) \rightarrow \text{AM} \ldots \]
  \[ \[] \text{in}(\_\_) \rightarrow \text{out}(\_\_) \rightarrow \text{DONE} \]

- A transducer produces a stream of results from a stream of inputs. It has a channel signature of the following pattern:
  \[ Tr = \text{in}(\_\_) \rightarrow \text{out}(\_\_) \rightarrow \text{Tr} \[] \text{DONE} \]

- A driver sends commands out into the world. It has a channel signature of the following pattern:
  \[ Dr = \text{out}(\_\_) \rightarrow \text{Dr} \[] \text{out}(\_\_) \rightarrow \text{Dr} \[] \ldots \]
  \[ \[] \text{out}(\_\_) \rightarrow \text{Dr} \[] \text{DONE} \]

Given such a notion of pure forms, checking whether a given ware and packager are compatible could be reduced to checking whether their pure forms are compatible. Exploring this notion of pure form and applying it to the problem of ware/packager compatibility is future work.)

If the system integrator’s intuition suggests compatibility, her next task is to establish a correspondence between the ware’s and packager’s channels by creating a channel map. At this stage, she may read the channel signatures and realize that there is no correspondence between the channels, which means that the ware and packager are incompatible. If necessary, she also may write additional coroutines to establish a correspondence between the ware’s and packager’s channels that cannot be established with a channel map.

Once the system integrator has established a correspondence between the ware’s and packager’s channels, the final stage is for the Ciao compiler to validate the compatibility between the ware and packager. Given the ware’s and packager’s channel signatures and the channel map that
establishes the correspondence between their channels, the Ciao compiler automatically checks compatibility.

A ware and packager are compatible if their concurrent execution cannot cause deadlock. To test this, the Ciao compiler creates a CSP model of the parallel combination of the ware and packager and uses the CSP model checker FDR [26] to check whether this model successfully terminates or deadlocks. (The details of this model check can be found in my dissertation [10].) Absence of deadlock in this CSP model implies absence of deadlock in the combined ware and packager, which is the basis of their compatibility. The soundness of this check relies on the accuracy of the channel signatures, which cannot be automatically verified. As a guard against human error, the Ciao channel library detects deadlock at run time, halting the component's execution upon detection.

The current weak link in this compatibility check is the possibility that the Ciao module’s code does not conform to its channel signature. Although checking this conformance is an undecidable problem, the Ciao compiler could follow the lead of safe languages, like ML and Java. Such languages circumvent undecidable typing problems, like array bounds checking, by instrumenting the compiled code with runtime checks. The Ciao channel implementation could similarly use an online algorithm to determine whether the trace of in and out statements is allowed by the channel signature. Such runtime checking is future work and could be done in the style of Colcombet and Fradet [8].

3.4 Flexible Packaging Overview

In summary, this section discusses the two key aspects of Flexible Packaging: the introduction of the role of the packaging specialist to prevent burdening the system integrator with packaging details and the use of channels to support the mix-and-match of packagers and wares. With these ideas established, I now provide an overview of the method as a whole. The role each participant plays is briefly described below.

3.4.1 The Ware Provider’s Job

The ware provider creates source code in Ciao to implement the functionality that he wishes to provide. As part of the ware’s development, he chooses those interactions that are flexible and those that are fixed. The flexible interactions are implemented with channels; the fixed interactions are implemented in the traditional way, for example, by calling I/O libraries. Choosing which interactions to flex and which to fix is an engineering decision, not a dictate of the method.

Once he has created his source code, the ware provider creates a channel signature that represents his code. As a leg-up on this task, he can run a tool called SigGen [10] that automatically generates a conservative approximation of the channel signature from the source code. As mentioned earlier, the problem of deciding whether a Ciao module’s source code conforms to its signature is undecidable. Similarly, automatically extracting a signature from the code is also undecidable. As an approximation, SigGen first produces a control-flow graph for each function definition in the source code and, for each function, translates the nodes to a signature as follows: in and out statements are translated to in and out process prefixes, a call to function F is translated to a “call” to the process F using the sequencing (semicolon) operator, conditionals whose predicates involve only calls to the stream function more are translated to internal choices, other conditionals are translated to external choices, and return statements and end-of-function nodes are translated to the process DONE. Because this generated signature is approximate, the ware provider must review it and may edit it to improve its accuracy.

As a last step, the ware provider makes the source code and channel signature available for use.

3.4.2 The Packaging Specialist’s Job

The packaging specialist is a guru about some particular packaging, like ActiveX. From his experience with this packaging, he decides which aspects of the packaging are essential and which are details. First, he uses the UniCon notation to describe the parameters of the packaging description that the system integrator must provide. This packaging description represents the essential decisions about the packaging. He then produces a generator that maps the system integrator’s packaging description to the source code, noncode artifacts (like a plug-in’s resource fork), and construction and installation steps needed to implement the described packaging. The Flexible Packaging method provides a Java framework that simplifies three common generation tasks [10]: parsing UniCon descriptions, creating Makefiles, and generating Ciao source code. The packaging specialist’s final step is to make the packaging generator available for use.

3.4.3 The System Integrator’s Job

After selecting a ware to use, the system integrator turns the ware into a full-fledged component in two steps, pictured in Fig. 5. The first step is to write a packaging description, which expresses how she wants the final component to interact with other components in the system. Examples of packaging descriptions are shown in Fig. 2b and c. She then invokes the packaging generator on the packaging description, which produces a packager with its channel signature, noncode artifacts, and construction and installation steps.

Her second step is to combine the packager and the ware. Having inspected both of the channel signatures and having read the ware’s and packager’s documentation, she finds an appropriate correspondence between the ware’s and packager’s channels. (If she cannot find such a correspondence, she realizes that the ware cannot sensibly be packaged according to the packaging description and can either reject the ware or rewrite the packaging description.) She records this correspondence between channels as a channel map. Finally, she runs the Ciao compiler on the ware, packager, and channel map, which produces the final component. This component is ready for integration in her system.
To evaluate the feasibility of using Flexible Packaging to develop “real-world” components, I performed a series of experiments. Playing the role of the ware developer, I developed three wares that implement three diverse pieces of functionality: image painting, data translation, and text classification. Playing the role of the packaging specialist, I created nine packaging generators that represent nine packagings used in practice today. Finally, playing the role of the system integrator, I packaged these wares to form 13 different components.

Fig. 5. The process a system integrator follows to produce a component. She selects a ware that implements the desired functionality and a packaging generator that handles the desired packaging. She writes a packaging description and runs the packaging generator on it to produce the packager, noncode artifacts, and construction steps. She creates a channel map to show the correspondence between the ware’s and packager’s channels and runs the Ciao compiler on the ware, packager, and channel map. The compiler in turn runs to produce the final component.

4 EVALUATION

To evaluate the feasibility of using Flexible Packaging to develop “real-world” components, I performed a series of experiments. Playing the role of the ware developer, I developed three wares that implement three diverse...
4.1 Experimental Wares

The wares that I created for the experiments are diverse, both in the domain of the functionality that they provide and in their use of channels to coordinate with packagers. Because I needed to spend the bulk of the validation effort on studying and encapsulating various packagings, I intentionally kept the wares simple.

4.1.1 Area Code Translation

In order to accommodate an ever increasing need for new telephone numbers in western Pennsylvania (USA), the 412 telephone area code was recently split into two area codes, 412 and 724. Whether a given phone number remained in the 412 area code or switched to the new 724 area code was determined by its exchange (first three digits). One of the effects of this change is that phone numbers must be updated in many databases and other electronic artifacts. The variety of artifacts to be updated is staggering: relational databases from a number of vendors, spreadsheets, formatted text files, text documents and document templates, web pages, electronic business cards, address books in contact managers, and many others.

Such a problem provides a natural opportunity to use Flexible Packaging. I created a ware that translates old phone numbers to new phone number and that is flexible about the source and sink of the phone data. The ware’s main value is in encapsulating the table of phone exchanges in the new 724 area code. This modest data transformation problem is representative of a large class of problems, some with enormous amounts of processing.

The area code ware is about 100 lines of Ciao and has the following channel signature:

```
convert = in(Phone) -> out(NewPhone) -> convert [] DONE.
```

channel in stream char* Phone
channel out stream char* NewPhone

The ware loops over the contents of the Phone stream, converting each phone number from the old area code to the new and placing it on the NewPhone channel. If a string from the Phone stream cannot be parsed, it is place unchanged on the NewPhone channel.

4.1.2 PNG Image Painting

The Portable Network Graphics (PNG) image standard was recently designed to be a successor to the popular GIF standard. One of the reasons the GIF standard still prevails is that many different kinds of software need to display images—drawing programs, document editors, stand-alone image viewers, user interface design tools, web browsers—and each imposes its own packaging requirements on the image-handling component. Creating a PNG viewing component for each of these niches takes time. Here, too, is a natural opportunity for Flexible Packaging. We would like once to capture the functionality of parsing and displaying a PNG image and reuse it in many different contexts.

The PNG ware is about 400 lines of Ciao and uses Randers-Pehrson, Dilger, and Schalnat’s libpng library for parsing PNG files. Its Ciao signature is the following:

```
PNG_Main = in(Init) -> PNG_Main
[] in(NewFile) -> PNG_ReadFile;
   PNG_Main
   [] in(Paint) -> PNG_Main
   [] in(Finalize) -> DONE.
PNG_ReadFile = out(ErrorMessage) -> DONE
[?] ConvertPNGToDIB; DONE.
ConvertPNGToDIB = out(ErrorMessage) -> DONE
[?] DONE.
```

channel in int Init
channel in struct { struct { long left, bottom, top, right; }*
    rect; void* hdc; } Paint
channel in char* NewFile
channel in int Finalize
channel out char* ErrorMessage

According to this channel signature, the ware first waits for an indication that it should initialize and then loops to handle commands. If it receives the name of a new file, it parses the file, possibly reporting a parse error. If it receives a painting context, it paints the last file that it parsed or a white rectangle if no file has yet been parsed. Finally, if it receives an indication to quit, it does so, after deallocating its resources.

The area code ware uses the absence of data on its input channel to determine when to terminate; hence, its input is a stream channel. In contrast, the PNG ware uses the presence of input on an explicit Finalize channel to determine when to terminate, and hence, does not use any stream channels. Stream channels provide direct support for the common case of simple data transformation problems; more sophisticated protocols are spelled out using the signature process.

4.1.3 Chat Message Threading

The members of the computer science community at Carnegie Mellon University use a chat system, called Zephyr, to share information ranging from the technical to the frivolous. The chat message display programs show the messages as a list ordered by their time stamps. Since there are typically several conversations on different topics occurring at the same time, this flat presentation makes it difficult to distinguish the conversations and to follow them independently.

To improve this presentation, I created a ware to break the stream of messages into distinct conversations. The ware is flexible both about the source of the Zephyr messages and about how the resulting conversations are consumed. Because the notion of conversation is not intrinsic to the Zephyr model, the ware uses heuristics both to attribute a message to a conversation and to determine when a new conversation is started. The ware is about 100 lines of Ciao and has the following channel signature:
The ware loops over the contents of the ZephyrNotices stream. For each message, it determines whether the message belongs to an existing or a new conversation. If it belongs to an existing conversation, the ware announces a new message belonging to that conversation. If it belongs to a new conversation, the ware first announces a new conversation and then announces the message.

### 4.2 Experimental Packagings

To experiment with a wide variety of packagings that are popular among today’s practitioners, the roles of nine different packaging specialists were played, each time creating a packaging generator for a particular packaging. Because many of these packagings are quite complicated and take years to master, becoming a true packaging guru nine times would have required an untenable level of effort. Instead, one needs to capture enough of the packaging’s complexity to be representative without spending more than a few months learning any one technology.

To support the experiments, packaging generators were created for each of the following Component Standard packagings:

- ActiveX controls,
- Netscape plug-ins,
- Windows applications (with graphical user interfaces),
- batch programs, and
- CGI scripts.

Also created were packaging generators for each of the following Data Access packagings:

- Excel spreadsheets, accessed through COM,
- relational databases, accessed through ODBC,
- text streams, and
- TCP/IP socket streams.

As mentioned earlier, for a component to have a given packaging often requires the creation of noncode artifacts and the invocation of packaging-specific construction and installation steps. Several of the experimental packagings involve these construction complexities. The table in Fig. 6 summarizes these construction complexities for those experimental packagings that have them.

### 4.3 Experimental Components

With the previous wares and packaging generators in hand, I created each of thirteen components by writing a packaging description for one of the wares. Some of the packagings were homogeneous (for example, ActiveX alone); others were heterogeneous (for example, both ActiveX and TCP/IP sockets). Fig. 7 lists the components, showing the wares and packagings from which they were built.

#### 4.3.1 Components without Wares

Four of the experimental components are unusual in that they contain no ware. One way to overcome a packaging mismatch is to interpose a bridge between a reused component and the new system in which the component is reused. Such a bridge encompasses no interesting functionality, but simply overcomes the differences between two types of interaction. Although the packaging generators were designed to produce packagers that complement wares, these generators can also be used automatically to produce such bridges.

A bridge component contains no ware (because it has no interesting functionality), but has a heterogeneous packaging. For example, component C10 bridges between socket-based interaction and ActiveX events. It listens for Zephyr messages on a socket and announces each incoming message as an ActiveX event. This component was created by writing a packaging description that involves both

---

**Fig. 6.** The noncode artifacts, constructions tools, and installation steps that the experimental packagings require.
sockets and ActiveX events and by creating a channel map that connects different channels in the packager (rather than between the packager and the ware).

### 4.4 Observations

The major result of the experiments is to establish that the Flexible Packaging method is capable of creating "real-world" components. In particular, Flexible Packaging handles both the complexity of packagings used in practice today and components with heterogeneous packagings. Beyond this, there are several experiences from the experiments worth reporting.

#### 4.4.1 Packagers with Multiple Threads

Several of the experimental packagers inherently require multiple threads of control. Consider component C8, the chat message ActiveX control, which is simultaneously an ActiveX control and a TCP/IP socket client. When one ActiveX control calls another control’s COM method, the caller gives its thread to the called control, in the style of remote procedure call. The called control is expected to perform its service, then return the thread back to the caller. If the called control were to hold the thread indefinitely, the caller could be left without any active threads, leading to such bad behavior as an unresponsive user interface. Hence, no COM method should compute indefinitely, for example, by containing a loop that executes for an arbitrarily long time. Component C8, however, needs exactly such a loop for listening to the socket. Since a message could appear on the socket at any time, processing the message soon after it arrives requires either blocking on the socket or frequently polling the socket. As previously mentioned, component C8 cannot block a caller’s thread, nor can it use a caller’s thread to poll since the caller would then determine the polling frequency. In short, component C8 needs an independent thread for listening to the socket. Thus, component C8 is inherently multithreaded: one thread arises from another control calling its methods; one thread is needed for blocking on the socket. Component C10, similar to C8, is the only other experimental component that is multithreaded.

This use of multiple threads of control is one solution to Garlan, Allen, and Ockerbloom’s multiple control loop problem [13]. As part of building a software architecture editor, they created a component that interacts through remote procedure calls, messages, and a graphical user interface. Each of these forms of interaction requires a control loop: a loop to listen for incoming remote procedure calls, a loop to listen for incoming messages, and a loop to handle user interface events. Because these loops must execute simultaneously, Garlan, Allen, and Ockerbloom revised the source code for these tools to combine the three loops into one. Using a different threads of control for each control loop solves the problem while maintaining the tools’ abstraction boundaries. This ability to combine control loops without inspecting their implementations is vital for Flexible Packaging, since these control loops are the product of packaging generators that independent packaging specialists create.

#### 4.4.2 Performance

One of the costs of using Flexible Packaging is the runtime overhead that the channel mechanism imposes. This is not a
fixed cost, but varies depending on the number of channel communications and the amount of computation performed between those communications. The more channel communications there are between the ware and the packager, the more runtime overhead the component will experience; the fewer computations performed between communications, the more runtime overhead the component will experience.

To measure this overhead, the three components from the area code case study were reimplemented to combine the code from the packager and ware into one module, removing the use of channels. The difference in execution time between the original and hand-altered versions of the components were then measured. Based on these measurements, each component experiences the following percentage of runtime overhead due to channels: 8 percent for the filter, 2 percent for the ODBC database accessor, and 1 percent for the Excel spreadsheet accessor. The variation is due to the different execution times of the three packagers. Taking consistent measurements for the three PNG components is infeasible because these components interact with the user.

4.4.3 Development Costs
The data in the experiments is too preliminary to use as the basis for quantitative cost estimates. However, we can make a qualitative comparison of the development costs of today’s practice versus that of Flexible Packaging.

Today, a component provider must know about both the domain of the functionality that he provides and the details of the component’s interaction style. By providing a ware rather than a component, he no longer needs to know about any particular interaction style; instead, he must know about channels, which are considerably simpler. Lowering the knowledge needed to produce a component lowers both component development cost and the barrier to entry into the reusable component market.

Today’s system integrators face two kinds of costs. When an integrator must overcome a packaging mismatch, she faces the direct cost of producing glue code or employing some other fix. When a component’s packaging does match the integration context, she faces the opportunity cost of disregarding those components whose packaging does not match. Flexible Packaging addresses both of these costs.

With respect to the direct cost, the experiments show that a given ware can be made available through a variety of packagings and, hence, can be reused in a variety of systems. The integrator’s direct cost of integration is typically the cost of producing a channel map and occasionally the cost of producing additional coroutines. The effort required to do this is less than that of producing glue code, in part because the former requires only knowledge of the channels involved while the latter involves detailed knowledge of two or more forms of component interaction.

Flexible Packaging also addresses the integrator’s opportunity cost. This opportunity cost increases with the number of components that the integrator neglects in order to avoid the cost of overcoming packaging mismatch. Further, the cost of overcoming a packaging mismatch with respect to a given component increases with the number of interaction commitments that are embodied in that component, since each commitment is a potential source of mismatch. Hence, the more interaction commitments that a component embodies, the more potential it has to impose a direct integration cost, and the more likely it is to be neglected to avoid this cost. Because wares embody far fewer interaction commitments than traditional components, they are less likely to be avoided and therefore impose less of an opportunity cost.

The packaging specialist has no real analogue in today’s practice and his role is the most expensive since he must produce a software generator. However, there need be only one packaging specialist per style of component interaction. Today, there are only a few dozen popular styles of component interaction and new styles of interaction are seldom introduced. Hence, few people will bear the expense of being a packaging specialist.

4.4.4 Unhandled Aspects of Interaction
There are three aspects of component interaction that Flexible Packaging currently does not handle well: failure (can a given interaction fail?), state persistence (across how many interactions should the component’s internal state last?), and state scope (how much of the component’s internal state should be modified because of a given interaction?).

First, Flexible Packaging does not insulate the ware from differences in interaction failure. In particular, in and out statements do not support a notion of failure. Consider a packager that implements interaction over a network with weak packet delivery guarantees. If the ware uses an out statement to provide a result to other components, the ware’s out statement receives no feedback about whether the result is actually delivered over the network. Similarly, if the ware does an in statement, the packager is expected to provide data through a corresponding out statement. There is no way for the packager to report a lack of data due to dropped network packets. For example, the packager must either provide data in a corresponding out statement or never return control back to the ware.

There are three strategies for coping with this deficiency, each with its own drawback. First, a channel that is allowed to fail can be supplemented with a second channel that passes out-of-band error information. This approach means that in statements on the possibly failing channel must be replaced with alt statements to allow the error channel also to be read. Second, the use of channels can be supplemented with other language mechanisms, like exceptions. The downside is that the channel signature would no longer completely describe a module’s interface, since it would not cover the use of exceptions. Third, the data value communicated on the possibly failing channel can be supplemented to include sentinel failure values (e.g., NULL). Unless the language’s type system is expressive enough to make the use of these values clear (C’s type system is not), then the use of sentinel values can cause type mismatches that are not evident in the channel signatures. Instead, Flexible Packaging should provide a direct way to describe the failure properties of a channel, which would be checked as part of ware/packager compatibility; this is future work.

Flexible Packaging also does not insulate the ware from differences in state persistence or state scope. The ware’s
state is private to the ware and the packager has no way to affect how long the ware’s state persists or how much of the ware’s state is affected through channels. An example of this is the PNG ware, whose internal state supports only a single PNG image at a time. This is an incompatibility with the Netscape plug-in packaging, which supports having multiple simultaneous instances of the plug-in, each with its own state. Hence, the PNG Netscape plug-in (component C6) behaves correctly only when a web page contains a single PNG image.

To correct these deficiencies, the ware could provide the packager with an abstract datatype that represents the ware’s internal state. To support changes in state scope, the abstract type would support create and destroy operations; to support changes in state persistence, it would support operations to convert to and from a persistent representation, like a byte stream. Updating Flexible Packaging to include these features is future work.

5 RELATED WORK

5.1 Separations of Concern

Gelernter and Carriero were early advocates of the separation of a software component’s functionality from its interaction with other components [15]: “We can build a complete programming model out of two separate pieces—the computation model and the coordination model. The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The coordination model is the glue that binds separate activities into an ensemble.” They created Linda as a coordination language to supplement a variety of popular programming languages (or computation languages, in their terminology) [6]. The Linda model is where computations interact through a shared persistent database of tuples, called a tuplespace. A given programming language, like C or Fortran, is supplemented with an out statement for adding a tuple to the tuplespace and an in statement for removing a tuple from the tuplespace. (I chose the keywords for Ciao in recognition of this earlier work.) Unlike Ciao communication, which is broken down by channels, the Linda tuplespace is global. However, because an in statement removes a tuple that matches a specified pattern, a programmer can use the patterns to simulate separate channels or to create any partitioning that he finds handy.

Like Flexible Packaging, Gelernter’s and Carriero’s work separates functionality from interaction. It advocates a sufficiently general interaction mechanism, shared tuple-space, that simulates other interaction mechanisms, like RPC and message passing. Rather than advocating a single interaction mechanism, Flexible Packaging allows the system integrator to select an interaction mechanism. Indeed, the ability to mix-and-match pieces of functionality and interaction mechanisms is Flexible Packaging’s main focus. This difference between the research projects in part reflects the research communities in which they arose. The Linda work arose in the parallel programming community, where component interaction allows large computational tasks to be decomposed into manageable pieces; Flexible Packaging arose in the software engineering community, where component interaction allows heterogeneous parts to be composed into systems.

Callahan’s thesis Software Packaging addresses a restricted case of the problem that Flexible Packaging addresses [4]. As with Flexible Packaging, the goal of Software Packaging is to allow a software component to interact through multiple mechanisms. Callahan restricts the range of mechanisms to variations of procedure call: local procedure call, remote procedure call, and cross-language procedure call. This restriction offers two advantages. First, because the range is of interaction mechanisms is narrow, there is no need for packagers. His analogue of a ware exports procedure definitions and uses, which can be readily attached to any of the three procedural connectors. Second, because the choice of mechanisms is fixed, he implements an inference system that automatically selects the appropriate procedural connector to use between any two components. Given declarative descriptions of the components, his engine selects remote procedure call, if the components are on different machines, cross-language procedure call, if the components are implemented in different programming languages, and local procedure call, otherwise.

Flexible Packaging is also similar to the work on Aspect-Oriented Programming (AOP) at Xerox PARC [19]. The goal of both research projects is to separate a component’s functionality from extrafunctional concerns. For Flexible Packaging, the extrafunctional concern is component interaction. AOP has explored several different extrafunctional concerns, including thread synchronization and data transfer in distributed programs [20]. In their approach, a program written in a standard programming language, like Java, is supplemented with a declarative description of an extrafunctional concern, called an aspect. For example, a Java class with internal data could be supplemented with an aspect that describes how threads should synchronize on that data. An aspect is expressed relative to the functionality, often including names that appear in the source code that implements the functionality. For instance, the thread synchronization aspect would include the name of the class and data items involved in the synchronization. Because an aspect is stated relative to the functionality, it is not reusable independent of the functionality. In contrast, with Flexible Packaging, a packaging description (in AOP terms, an aspect about component interaction) is an independently reusable artifact and is not expressed relatively to the functionality. The price to be paid for this reusability is that the system integrator must create a channel map to show the relationship between the functional and extrafunctional concerns and their combination unfolds at run time as the ware and packager use channels to coordinate. In contrast, because aspects are expressed relative to the functionality, at compile time the AOP compiler weaves the source code that implements the functionality together with the source code generated from the aspect.

5.2 Concurrency and Modularity

The benefits of concurrency on program structure have been studied for a long time. Conway invented coroutines in the early 1960’s, for example, to provide a good structure
for his cobol compiler [9]. Like typical modern compilers, his compiler's modules implemented the various phases of compilation, like lexing and parsing, each of which can be seen as a state machine. The concurrency inherent in the coroutine mechanism allowed these machines to proceed independently of one another.

Kahn and MacQueen report their experience using networks of processes that communicate through data channels (later known as Kahn-MacQueen networks) to structure programs that, for example, sort, compute power series, and compute Fourier transforms [18]. Kahn and MacQueen separate the programming model from the execution platform. A network of processes can execute on a single processor with the channels implemented as coroutines or on a multiprocessor.

A similar notion of process networks underlies the Jackson System Development (JSD) method, in which a system is developed in three phases: in the model phase, the developer uses processes and events (data communicated asynchronously on channels) to describe the parts of the world that circumscribe the problem the system is to solve; in the network phase, the developer adds processes that describe the system's solution to the problem; in the implementation phase, the developer maps the resulting network of concurrent processes onto the available computing and data storage resources [5]. With JSD's primary focus on the events that processes exchange, Jackson argues that JSD is better suited to describing systems that change over time than older approaches, like entity/relationship diagrams.

More recently, Reppy argues that concurrent languages are useful for structuring interactive, distributed, and reactive systems: “These applications share the property that flexibility in the scheduling of computation is required. Whereas sequential languages force a total order on computation, concurrent languages permit a partial order, which provides the needed flexibility.” [24] His language Concurrent ML (CML) has been used to create a user interface toolkit based on the X protocol and a distributed programming toolkit [25].

In all of these works, the use of concurrency to promote good program structure arises in the nature of the problem. Instead, with Flexible Packaging, the use of concurrency to promote good program structure arises in the nature of the development process. Reiterating Reppy's point, switching from a sequential to a concurrent programming notation allows developers to specify a partial order of computation rather than a total order. In the previous examples, this flexibility in the order of computation allows the system to implement the flexibility inherent in the problem (e.g., the flexible order of operations in an interactive system). With Flexible Packaging, the added flexibility is used to foster compatibility between independently authored modules. With Flexible Packaging, the variability in the order of computation arises not only because of variable runtime phenomena (like the order of user operations) but also because of variable combinations of Ciao modules.

### 5.3 Channel Signatures and CSP

The use of CSP in Ciao channel signatures was suggested by Allen's use of CSP in the Wright architectural description language [1]. In Wright, a system is described as a configuration of components and connectors. Components provide the system's functionality; connectors mediate the interaction among components. A component description in Wright captures how the component interacts with the “outside world” to provide its functionality. The description contains a port for each kind of interaction in which the component participates and a computation that captures how all the ports' interactions are combined to form the component's total behavior. Both ports and computations are specified as CSP process definitions. Just as a component's interactions are factored into ports, a connector description is divided into roles. Each role describes how a component taking part in that interaction must behave. A connector description also has a glue description that captures how the roles' interactions are combined to form the connector's total interactive behavior. Wright checks both the internal consistency of individual component and connector descriptions and global properties of the configuration, like deadlock freedom.

Channel signatures can be thought of as simplified Wright descriptions. A channel signature captures a Ciao module's interactive behavior in the same way that a Wright component's computation section captures the component's interactive behavior. Indeed, a channel signature can be thought of as the Wright component description of a Ciao module, with the syntax simplified and specialized. Given this, a plausible approach to checking the compatibility of Ciao modules would be to use Wright directly. We could create a Wright description of a configuration that consists of the ware and the packager described as components and Ciao channels described as connectors. We would then run the Wright tools on the configuration, which would in turn run FDR. Since the full generality and power of Wright are already hidden from Ciao users (in the name of ease of learning), the Ciao compiler instead skips the middleman and invokes FDR directly.

### 6 Future Work

The software architecture community, over the past several years, has invented notations for system composition, called architectural description languages (ADLs). A typical ADL describes a system as a configuration of components and connectors: Components house the system's functionality and connectors mediate the interaction among components [30]. As mentioned in Section 1, today's software components embody commitments both about functionality and about interaction with other components. The component/connector split in ADLs reflects this practice. A description of a component in an ADL captures the commitments that the component makes about interaction. A component might be described as a Unix filter interacting through text streams, as a procedure library interacting through procedure calls, or as an internet server interacting through a request/reply protocol. A component's description determines the eligible connectors to which it can be attached to form a system. A Unix filter component, for example, can be attached to a pipe connector, but not a procedure call connector or a socket connector.
connector. One of the main achievements of ADLs is the creation of type systems that capture the connectivity limitations of current practices.

Because ADLs reflect current practice, the component/connector separation does not coincide perfectly with the functionality/interaction separation. Although connectors in ADLs are purely about interaction, components are about both functionality and interaction. The Flexible Packaging method has taken the first step in bringing these two separations into alignment. Using Flexible Packaging, developers can create components (wares) with commitments about functionality, but very few commitments about interaction. Flexible Packaging advocates a process for the system integrator that is the least disruption from current practice. The system integrator first gives a ware a concrete packaging, turning it into today's notion of a software component. Then, with a familiar software component in hand, the system integrator can follow current practice, namely, composing the component with connectors to form the final system.

To improve the Flexible Packaging approach, a new ADL in which the component/connector separation coincides with the functionality/interaction separation will be explored. The evolution of this new ADL can be pictured as follows:

The first step represents today's practice, where a system is composed from components (ovals) and connectors (diamonds), which are attached (black squares) to form a system. The components themselves intermix the concerns of functionality (white) and interaction (gray). Notice that the component/connector distinction (at the black squares) does not coincide with the functionality/interaction distinction (at the gray/white border). Flexible Packaging takes the next step: A developer creates a component by combining a ware, which captures functionality, and a packager, which captures interaction. Other than its implementation as a packager and a ware, a software component is otherwise conventional. The two separations still do not coincide. The third step, to be taken in future research, is to associate the packagers, not with the components, but with the connectors. With this regrouping accomplished, the component/connector boundary and functionality/interaction boundary coincide.

There are several possibilities that this final step creates. First, one could create a rapid prototyping environment for component interaction. In current software architecture editors, like those for UniCon and ACME [14], an architect creates a system description by attaching components and connectors, which are depicted as box-and-line diagrams. These attachments are constrained by the typing restrictions previously mentioned: filters can only be connected with pipes; procedure-based modules, with procedure calls; etc. In this new environment, the components would be wares, and the connectors would encapsulate everything about a given type of interaction, including the packaging generators. This means that an architect can easily try different connectors between the wares to try out different forms of interaction. For example, if the two wares need to share data, an architect could insert a database connector between them, which provides noninterference properties at the expense of slower data access times. If the database's performance were too slow, he could change this connector to a shared memory connector, which provides quicker data access but provides no guarantees about the absence of interference. Because the commitments about interaction have been moved out of the components and into the connectors, connectors can be more readily substituted one for the other than they can in current architecture editors. Further, such an environment could infer or provide guidance about the proper connector to select. For example, if the architect's goal is to place a connector between one ware's in channel and another ware's out channel, a unidirectional dataflow connector, like a pipe, would be appropriate.

7 CONCLUSION

This paper introduces the Flexible Packaging method, which allows a component's functional and interactive concerns to be separated. A component's functionality is captured in a ware. A ware's use of channels allows it to specify enough about interaction to express its functionality, while leaving most details unspecified. A component's interaction is captured in a packager, which is automatically generated from a high-level description of the component's packaging. This method supports the following reuse scenario: A system integrator takes a ware "off the shelf" with the functionality that she needs, describes the packaging it must have to be compatible with the system's architectural style, and then uses the Flexible Packaging tools to turn the ware and the packaging description into a software component with the specified packaging. This tailored software component can then be directly integrated into the system.

ACKNOWLEDGMENTS

The author is grateful to Mary Shaw for her timely, helpful feedback on this paper and for suggesting the area code case study. The Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, US Air Force, and the Advanced Research Projects Agency have supported this work under grant F33615-93-1-1330. This paper represents the views of the author and not of Carnegie Mellon University nor of any of the sponsoring agencies. Microsoft, ActiveX, and Windows NT are trademarks of the Microsoft Corporation. Netscape is a trademark of the Netscape Corporation.

REFERENCES


Robert DeLine recently graduated from Carnegie Mellon University, where he completed a dissertation on software component interoperability. Now at Microsoft Research, he is working on a new programming language for creating modules that interact according to checkable rules. He is a member of the IEEE.