Robert DeLine

Resolving Packaging Mismatch

Submitted in partial ful llment of the requirements for the degree of Doctor of Philosophy

CMU-CS-99-141

The following agencies sponsored the research in this dissertation: the Defense Advanced Research Project Agency (DARPA) under contract N66001-95-C-8623; the United States Air Force (USAF) and Rome Laboratory under contract F30602-97-2-0031; the USAF, Wright Laboratory, and Air Force Materiel Command under contract F33615-93-1-1330; and the National Science Foundation (NSF) under grant CCR-9357792. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University, the United States government, or any of the sponsoring agencies.

Abstract

To integrate a software component into a system, the component 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. As a result, system integrators often face mismatches between their system's interaction commitments and those of a component to be integrated. How a component interacts with other components is called its *packaging*; hence these mismatches are called *packaging mismatches*. Packaging mismatches are typically resolved by introducing so-called glue code, which is costly to produce and maintain.

In this dissertation, I introduce a software development method, called Flexible Packaging, which allows a component developer to defer many decisions about component interaction until system integration. Using this method, a system integrator first selects a component, called a *ware*, with few interaction commitments and then writes a *packaging description*, which captures how the ware is to interact with other components. Given a ware and a packaging description, the Flexible Packaging system automatically produces the source code, non-code artifacts, and construction and installation steps needed for the component to exhibit the described form of interaction. By writing a packaging description, the system integrator tailors the component's interaction to the system at hand, and the component can be readily integrated. I have validated the feasibility of the Flexible Packaging method and its tools with several case studies, which involve a wide range of interaction: COM servers (ActiveX controls); COM clients; Netscape plug-ins; Windows applications with GUIS; ODBC-based database accessors; filters; CGI scripts; and TCP socket clients.

Contents

Acknowledgements

The fundamental success factor in earning a PhD is always the advisor/student relationship – a rule for which my experience is no exception. When I first began working with Mary Shaw, meeting with her was like swimming in the Sea of Abstraction. I'd take a deep breath, dive under, and enjoy the sights for as long as I could. But my terrestrial need for the Concrete would eventually send me back to the surface. Over time, with Mary's patient training, I could stay down longer and longer to hunt for pearls. The skills that I've learned from Mary – finding the real problem buried in the domain details, applying results from seemingly different fields, distilling my own experience to guide others – will sustain me for the rest of my career. For these skills I am truly grateful to her.

Skills alone, however, would never have gotten me through the program. There were many times when I felt discouraged, if not incompetant. I mistakenly thought that learning to do research is like learning to drive or to cook – that after an initial phase of learning and practice, the ability would become routine, second-nature, effortless. Mary's nicest gift to me was to dispel this myth: at no point does research become easy; struggle is part of the job. For this lesson, too, I am deeply thankful. Thanks also to my friends who helped me through the discouraging times, especially to Kip Walker and Sasha Wood, without whom I simply never would have made it.

The inherently solitary nature of PhD research often made me long for the esprit de corps that comes from teamwork. Many thanks to Greg Zelesnik for providing that spirit. Creating demos with Greg for our sponsors and customers was truly a lot of fun. Thanks also to Greg and to Margaret Weigand for insulating me (and many others) from the more mundane aspects of keeping our sponsors happy. I much appreciate the frustrations they bore on behalf of everyone. I am, naturally, also grateful to the sponsors themselves, who enabled the work altogether.

Finally, I could never have done this research in the absence of great colleagues. My thanks to Gregor Kiczales for providing the summer internship that proved so pivotal. My exposure to his group's ideas was just the kick in the pants I needed to find a thesis. I am also grateful to Daniel Jackson and to David Garlan for helpfully commenting on my ideas, whether half-baked or utterly raw. (Thanks also to Daniel for encouraging my typography fetish and for providing excellent examples to follow.) Big thanks to Bob Monroe for logging so many whiteboard hours with me and for thoughtfully considering even my dumbest ideas. Thanks also to the Software Systems Study Group and to our colleagues from the Software Engineering Institute for many lively discussions.

RAD

Every program is a part of some other program and rarely   ts.

ALAN J. PERLIS

1 *Introduction*

Software developers have long wanted to build systems from reusable parts. At the 1968 NATO workshop for which the term "software engineering" was coined, McIlroy gave a keynote speech that called for a market of "software components," his forward-looking name for routines.[33] He noted that "software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors." Inspired by this analogy with assembly line production, reuse enthusiasts reason that assembling systems from well tested, off-the-shelf parts reduces both development time and the cost of system development and validation.[5]

Unfortunately, this dream of building software systems from components remains largely unrealized, in part due to technical barriers. Putting aside the difficulty of finding and acquiring components,[54] a host of problems can arise when a developer assembles components into a system. One problem is that a component's functionality may be similar to the desired functionality, but insufficient in some regard. For example, Garlan, Allen, and Ockerbloom, in a study on building systems from reusable parts, needed to create a new transaction mechanism for their system because the off-the-shelf database that they used did not support sharing transactions across multiple address spaces.[19] Similarly, when Sullivan and Knight used a commercial drawing package to build a fault tree analysis tool, they found that the drawing package announced the addition, but not the deletion, of shapes. This missing functionality made it difficult for them to keep the internal state of the fault tree analysis consistent with the fault tree's depiction.[50] It is also not uncommon for a component's functionality to differ significantly from its documented functionality. This can mean that a component that appears on paper to be suitable may in fact be unusable.

Other problems arise during system assembly. One such problem is that components that perform well in isolation may together overtax a shared resource. In the same study, Garlan, Allen, and Ockerbloom found that their system, assembled from several libraries with large memory footprints, exhausted their workstation's memory resources. Similarly, a system assembled from several computationally intensive components might overwhelm the processor. A more subtle problem is semantic mismatch, which arises when components to be integrated disagree about the exact meaning of a shared concept. In an example due to Al Despain, several military inventory databases are united into an aggregate database. The inventory databases differ both in how items are counted (single items versus units of items) and on how the number of items "on hand" is counted (the number in stock versus the number in stock minus the number committed). Such semantic mismatches are often not discovered until the databases are united and produce bogus inventory counts.

Finally, components may be difficult to integrate because they differ in how they exchange data and control with one another. For instance, a Unix filter cannot readily interact with a procedure. A filter incrementally accepts data from an input text stream, computes over them, and incrementally reports its results to an output text stream; whereas, a procedure accepts data once through the procedure arguments, computes over them (with no constraint on access order), and reports its results once through the procedure return value. A filter and a procedure differ both in the mechanics of the data exchange (text streams versus the run-time stack and registers) and in the overall data access protocol (incremental versus batch); hence, they will not readily interact with one another. The style in which a component interacts (exchanges data and control) with other components is called its *packaging*; hence this last problem is called *packaging mismatch*. Packaging mismatch and techniques to resolve it are the subject of this dissertation.

To understand how packaging mismatch arises in practice, one must examine the two roles that are played in a software component's development and deployment: the *component provider* develops a component and makes it available for use; the *system architect* acquires components and assembles them to form a system. These roles may be played by the same person, by different members of a development team, or by two people separated in time and space. The heart of the packaging mismatch problem is that the participants playing both of these roles choose a component's packaging, which allows the possibility of conflict in their choices. This insight is the foundation for a new approach to component development and deployment, called Flexible Packaging.

1.1 A COMPONENT PROVIDER MAKES PACKAGING CHOICES

At the time a component provider develops a component, he chooses how it interacts with other components. He may make this decision by appealing to a component standard, which determines a given type of interaction. For example, he may choose to package his component as a procedure library (interaction through procedure calls), a batch system (interaction through file access), an ActiveX control (interaction through remote procedure calls), a filter (interaction through text streams), or an internet server (interaction through TCP sockets). Additionally, he may choose different sources of data over which the component computes, for example, a relational database, an event, a message, or shared memory.

Although technology exists to allow the component provider to defer some decisions about the component's interaction (discussed in Chapter 2), a component provider typically makes all decisions about a component's interaction at the time he develops the component. There are two key reasons for this. First, making at least some of these decisions at development time is inevitable given today's technology: to invoke a component's functionality, either for unit testing or for integration, requires the ability to interact with the component through some means. Second, choosing a particular packaging for a component is also necessary for targeting component markets. A component provider interested in selling to Visual Basic developers, for example, will package his component as an ActiveX control; one interested in selling to average consumers will package his component as an application with a graphical user interface.

The choices that a component provider makes about interaction are often difficult to change because they affect both the component's source code and its construction steps. The effects on the component's source code range from localized to pervasive. For example, a decision about a component's data source is often expressed as calls to procedures in an input/output library, like a text stream library, a socket library, or a relational database library. Such localized choices can often be encapsulated behind an abstract I/O interface, as was done, for example, in the well known A-7 project.[40] Other choices about interaction are pervasive, affecting for example the component's overall control structure. For instance, when the number and order of interactions is relatively constrained and known at development time, the code is typically organized using the "internal control" paradigm: the source code spells out the number and order of interactions through its sequencing of statements. When the number and order of interactions are relatively unconstrained, as is often the case with event or message interactions, the code is typically structured using the "external control" paradigm: the source code associates computations with the different kinds of interactions and then yields control to a dispatch mechanism. Whether the developer chooses the internal or external control paradigm affects the overall structure of the code. Further, the choice of interaction may allow the developer to use packaging-specific tools to develop parts of the source code. For example, a developer may use a parser generator like Yacc to implement text stream interactions; a stub generator, remote procedure call interactions; and a graphical user interface builder, event interactions. In short, choices about a component's interactions can have profound implications for its source code.

In addition to the source code, choices about a component's interactions also affect its construction and installation steps. For example, in order for a component to be packaged as a Netscape plug-in, the following construction and installation steps apply: the component's source code must be compiled into a dynamic library; a resource file with two particular text resources must be compiled (using a resource compiler) and associated with the dynamic library (through a linker option); the library must appear in the "plugins" subdirectory of the directory containing the Netscape application; and the library's name must be in DOS 8.3 format and begin with the letters *NP*. Although byzantine, these rules contain elements typical of many packagings,: the production of non-code artifacts (the resource file); the use of packaging-specific tools and tool options; and need for packaging-specific installation steps after compiling and linking. Hence, a component provider's choices about packaging affect not only the content of the component's source code, but how that source code is processed to form the final component.

Because the component provider's decisions about interaction affect both the component's source code and its construction and installation steps, these decisions are often difficult to change, especially once the component is made available for deployment. If the component is available as compiled code, then its packaging-specific construction process ossifies the component provider's decisions, rendering them unchangeable. When the component is available as source code with a construction script, as it is before deployment, the pervasive affects of some interaction decisions make changing these decisions difficult. Even when the decisions about interaction could be encapsulated within, say, an abstract input/output library, such encapsulation is rare in practice; the portions of source code that achieve the component's functionality are commonly intertwined with those that achieve the component's interactions.

In summary, a component provider typically makes most or all of the decisions about a component's interaction when he develops the component. After development, such decisions are difficult to change.

1.2 A SYSTEM ARCHITECT MAKES PACKAGING CHOICES

To help distinguish the various development roles throughout the dissertation, I use feminine pronouns for the system architect and masculine pronouns for the component provider and, later, the packaging specialist.

Historically, two terms have been used to describe a software developer who builds a system from parts. When the developer oversees the entire system and has free reign over system-wide decisions, she is traditionally called a *software architect* or *system architect*; when she builds the system from pre-existing parts, so-called legacy components, and is therefore saddled with many previously made commitments, she is traditionally called a *system integrator*. This historical contrast represents two ends of a spectrum. Whether she has many or few previous commitments to contend with, the system architect's job is to acquire software components (by building or buying them) and to assemble them to form a system. Hence I will use the same term, *system architect*, to describe this role, wherever it falls along this spectrum.

A key decision that the system architect faces is the selection of the system's architectural style, which determines how the components interact with one another. Several factors influence this choice. First, she may choose an architectural style for its influence on such overarching system properties as performance, security, and reliability. Consider, for example, the task of integrating several components that share data. Selecting a relational database to house the data provides specific guarantees against mutual interference among the components at the cost of relatively slow access to the data. In contrast, selecting shared memory to house the data provides no guarantee against mutual interference but allows relatively fast access to the data. Choosing the system's architectural style allows the architect to make trade-offs among global system properties.

A second factor is the system architect's personal experience or the collective experience at the architect's development organization. Many architectural styles involve specialized skills, like database schema design or security protocol design, and detailed knowledge, like understanding the use of packaging-specific tools. Once an architect has made the investment of acquiring the skills and knowledge associated with a given architectural style, she may sensibly opt to amortize the cost of that investment over the construction of many systems in that style.

These first two factors come into play when the architect is relatively unconstrained by inherited commitments. When the system architect uses legacy components, these components also influence her choice of architectural style. As discussed earlier, components are typically committed to a particular style of interaction, and these commitments are difficult to change. As such, the architect may choose to let these commitments drive her choice of architectural style. For example, a system architect's customer may insist that she use the customer's existing relational database in the system to be built. Because of this constraint, the architect may choose to use a database-centered architectural style. In general, each legacy component to be integrated places its own contraints on the architectural style, and the architect seeks a style that satisfies the constraints, if one exists. In practice, an architect's desired system properties, experience, and legacy components may all influence her choice of architectural style.

Once a system architect has selected an architectural style for her system, the components incorporated into the system must be reconciled with this choice. Different strategies for ensuring that the components adhere to the architectural style come at different costs. One strategy is for the architect herself to oversee development of the components to ensure that they use the necessary interaction mechanism. This means that the architect and her development team incur the components' development cost.

Another possibility is for the architect to select components only from a pool of candidate components that adhere to the architectural style. This is the basis of today's component market. For instance, if a system architect selects the Corba architectural style for her system, then she may choose to shop for components only in the Corba market to ensure compatibility. The cost of this strategy is the opportunity cost of shopping only within a niche market. Using this strategy, an architect committed to Corba will miss the opportunity to use COM or JavaBeans components that have the functionality she needs. The dual of this strategy – limiting the selection of an architectural style to those styles that are consistent with the components to be used – incurs an analogous opportunity cost at the architectural level.

A final strategy to ensure compatibility between a system's components and its architectural style is to select components based only on their functionality and then to resolve whatever packaging mismatch results. Although there is a spectrum of techniques to resolve packaging mismatches (which I catalogue in Chapter 2), two techniques dominate. The first is to re-implement the component to have the necessary packaging. As previously explained, identifying and isolating those parts of the source code and construction and installation steps that need to be changed can be quite difficult, which can make this an expensive option. The second technique is to interpose a new component that overcomes the mismatch between the reused component and the rest of the system. Such components are often referred to as wrappers, adaptors, mediators, bridges, or glue code. For example, a system architect committed to Corba may decide to use a COM component, introducing a COM/Corba bridge to overcome the mismatch. To produce such glue code requires a full understanding of the interaction mechanisms involved, which can be a considerable amount of knowledge. For instance, both COM and Corba are complicated mechanisms on their own, and a full understanding of both is needed to build a COM/Corba bridge. Even when the architect avoids the learning cost by using off-the-shelf glue code, the glue code adds new complexity to the system (for example, new points of failure) and often degrades system performance. Further, like any other system component, the glue code must be maintained over the system's lifetime. Hence, the glue code technique, too, is often an expensive option.

In summary, when a system architect builds a system from reused parts, the benefits of reuse come at a cost: either the opportunity cost of limited choices or the direct cost of resolving packaging mismatch.

1.3 TWO PARTICIPANTS, ONE DECISION

Fundamentally, packaging mismatch arises because two participants in the system development are making the *same* design decision, namely how a given component in the system interacts with other components. The component developer makes this decision at the time he develops the component. The system architect makes this decision when he selects an architectural style for his system, which determines how the system components will interact.

To the extent that their decisions agree, packaging mismatch is avoided. To ensure this agreement, either the component provider can accept the system architect's decision (the top-down or historic "system architect" approach) or conversely the architect can accept the component provider's decision (the bottom-up or historic "system integrator" approach). This agreement is most readily accomplished when the component provider and the system architect are the same person or members of the same team, namely when there is the opportunity for explicit collaboration between them. However, in our desired scenario in which a system architect builds a system from reused parts, the component provider and system architect may in fact work for different companies at different times, which prevents explicit collaboration between them. In the absence of this collaboration, the system architect may either limit her choice of components to those that agree with his choice of architectural style (the niche market approach) or may limit his choice of architectural styles to those that agree with the component providers' decisions (the legacy component approach). In short, to avoid packaging mismatch, today's software development technology either requires up-front collaboration between the component provider and system architect or limits the system architect's choices.

1.4 A NEW APPROACH TO COMPONENT DEVELOPMENT AND ASSEMBLY

This dissertation introduces a new software development method, called Flexible Packaging, which neither requires up-front collaboration between the component provider and the system architect nor severely limits the system architect's choice of components or architectural styles. Because the packaging mismatch problem stems from two participants making the same decision, the fundamental strategy of Flexible Packaging is to reduce the number of decision makers from two to one, thereby eliminating the potential for conflicting decisions. To do this, broadly speaking, the Flexible Packaging method takes decisions about a component's interactions out of the hands of the component provider and puts them into the hands of the system architect. The remainder of this chapter makes the nature of this decision making more precise.

As previously mentioned, when a component provider uses current technology to develop a component, he chooses the component's packaging, which affects both the component's source code and its construction and installation steps. In order for the Flexible Packaging method to take these packaging decisions out of his hands, the method supports a separation between a component's functionality and its packaging. With these concerns separated, the component provider makes the decisions about the component's functionality; the system architect, those about its packaging.

However, a component's functionality and packaging are not orthogonal concerns. A component's functionality affects the interactions in which the component participates and vice versa. For example, to achieve its functionality, a component performs some computations. Unless it performs those computations over a fixed data set, the data over which it computes must arise through interaction with other components, whatever form that interaction takes. Hence a component's functionality influences the interactions in which it takes place. Conversely, if a component is to support some standard form of interaction (like painting to a window), then its functionality must support that interaction (by implementing the painting). Hence a component's interactions influence its functionality.

Because of the mutual influence between functionality and packaging, these concerns cannot be wholly separated. To address this, the Flexible Packaging technique distinguishes essential decisions about a component's interactions from incidental decisions. In particular, the computation that implements a component's functionality must be able to accept the input over which it computes and to report results as it produces them. When the computation needs input (and of what kind) and when its results are ready (and of what kind) are intrinsic to the functionality. Hence, the number, types, and order of inputs and outputs that the functionality consumes and produces constitute the essential decisions about interaction; the mechanics of how the component interacts with other components to acquire the needed input and to report the produced output are incidental. Given this distinction, the Flexible Packaging method couples the essential decisions about interaction with the decisions about functionality, but decouples the incidental decisions about interaction. Hence, to clarify an earlier remark, the Flexible Packaging method takes the *incidental* decisions about a component's interactions out of the hands of the component provider and puts them into the hands of the system architect. Because of this distinction, in the remainder of the dissertation I use the word *packaging* to refer only to the incidental decisions about interaction.

Because of the mutual influence of functionality and interaction, it is impossible to achieve perfectly the strategy of reducing the number of packaging decision-makers from two to one. Because the component provider makes the essential decisions about interaction and the system architect makes the incidental decisions, there remains the possibility that their decisions will be incompatible. As described in Chapter 3, a major thrust in the design of the Flexible Packaging method is to maximize the likelihood of compatibility between their decisions and to ensure that this compatibility can be automatically checked.

In summary, here is the claim that this dissertation demonstrates:

It is possible to separate a software component's functionality from its packaging so that the following properties hold:

1 **The system architect can independently choose a software component's functionality and packaging at the time the component is to be integrated into a system.**

2 **She can readily determine whether her choices of functionality and packaging are compatible.**

3 **If her choices of functionality and packaging are compatible, she can automatically produce a software component based these choices. This component's performance is comparable to the equivalent component developed by hand.**

4 **Her functionality and packaging choices are compatible often enough to allow her to produce a useful variety of components.**

5 **The component provider and system architect expend less development effort on the whole using this approach than they would using traditional development methods.**

Given a development method that exhibits these properties, the system architect is able to select an architectural style for her system, then to choose a packaging for each component that is consistent with this style. Because an off-the-shelf component produced with this method is relatively free of packaging decisions, in many cases the system architect is able to tailor the component to the interaction requirements of her system. She no longer has to limit herself to packaging-based niche markets nor to create glue code to shoehorn components into the system.

1.5 A DEMONSTRATION OF FLEXIBLE PACKAGING

The remainder of the dissertation discusses the Flexible Packaging method in detail. To introduce the method in brief, consider a programmer building an application in Microsoft's Visual Basic. Visual Basic is a development environment geared toward the reuse of software components called ActiveX controls. Today, Visual Basic programmers represent one of the few communities in which software reuse regularly and systematically occurs. Briefly, an ActiveX control is a library exporting a collection of interfaces, called COM interfaces, each of which is a collection of procedures and properties. Many COM interfaces are standard and represent such common capabilities as windowing, painting, data exchange (copy and paste), and interface reflection; other COM interfaces represent control-specific capabilities. Today's most common ActiveX controls implement user interface elements, like buttons, sliders, and dialog boxes; other controls are utility libraries for common tasks, like mathematical calculations, calendar management, or HTML parsing.

For an example repository of ActiveX controls, see *www.activex.com*.

The Visual Basic programmer in our scenario has a picture of her corporate logo in the PNG image format. To insert this image into a dialog box for her application's "About…" menu item, she needs an ActiveX control capable of parsing and painting images in the PNG format. In a world in which the Flexible Packaging method is widely used, she performs the following steps. First, she needs a software component that has the desired functionality and is free of packaging commitments; such a component is called a *ware*. Her first step is to search software repositories, similar to those on the web today, for a ware capable of handling PNG images. Having found one, she inspects its *channel signature* and documentation to learn about its capabilities.

The PNG image format is a technical successor to the popular GIF image format. For details, see *www.cdrom.com/pub/png*.

A channel signature summarizes and abstracts the interactions in which a ware participates and forms a basis for describing the services that the ware provides. In this way, a channel signature is analogous to an interface in such module-based languages as Standard ML, Modula 3, and Ada or such interface description languages as Microsoft's IDL, which describes COM interfaces. A channel signature represents the ware provider's essential decisions about interaction and consists of two parts: a list of channels through which the ware interacts and an expression summarizing how the ware uses those channels. The following is the channel signature of the PNG ware that the Visual Basic developer finds:

This and the subsequent packaging description have been simplified to ease discussion in this introductory chapter. The complete versions of these signatures are shown in Chapter 5.

```
Ware =  in(Initialize) → Loop.
Loop =  in(NewFile) → Loop
     [] in(Paint) → Loop
     [] in(Finalize) → DONE.

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

Each channel in the channel list has a parity (*in* or *out*), a type (in the type system of the C programming language), and a name (a legal C identifier). For example, the first channel has parity *in*, type *int*, and name *Initialize*. A ware uses *in* channels to request the data over which it computes and *out* channels to report the results of its computations.

The subset of CSP consists of process prefixing (→), internal (non-deterministic) choice among processes ($\sqcap$), external choice among processes ([]), and the successfully terminating process (SKIP). In the syntax of channel signatures, internal choice is written with a question mark (?), and SKIP is called DONE.

The first part of the channel signature is an expression in a subset of Hoare's Communicating Sequential Processes (CSP) notation.[25] This expression describes the possible orders in which the ware might request or report data over the channels. The description of a ware's behavior always begins at the first rule, in this case the rule Ware. This rule states that the PNG ware first requests data over the channel Initialize and then behaves according to the second rule, Loop. The rule Loop states that the ware is willing to accept input on any one of three channels: NewFile, Paint, or Finalize. The use of square brackets ([]) to record this conditional means that the choice of which channel the ware will use is left up to the "outside world" rather than the ware itself. (The computation that constitutes the "outside world" is discussed shortly.) These so-called external choices are common among wares that provide services. The PNG ware can be thought of as providing three services – parsing (NewFile), painting (Paint), and quitting (Finalize) – which may be invoked in any order. (As it happens, if the painting service is invoked before the parsing service, the ware paints a while rectangle.) According to the rule Loop, if the ware receives input over the channel NewFile, it subsequently behaves according to the rule Loop. The same is true for the channel Paint. Finally, if the ware receives input on the channel Finalize, it successfully terminates, as the keyword DONE specifies.

This CSP description specifies the interactions in which the ware participates, but is silent on what computations are performed around those interactions. For example, although the ware presumably paints pixels to the screen after receiving input on the channel Paint, such functional behavior is not included in the channel signature and would have to appear in the accompanying documentation.

After the Visual Basic developer acquires the PNG ware, she writes a *packaging description* that specifies the packaging that she would like the ware to have. Because she is working in Visual Basic, she would like the

PNG ware to be an ActiveX control and therefore writes the following packaging description:

```
PNGViewerControlInterface: interface ActiveXControlPackaging with
   library_iid: UUID{7cf18aa0-36ff-11d2-9fd5-00104b33709d};
   coclass_iid: UUID{7cf18aa1-36ff-11d2-9fd5-00104b33709d};
   name: "PNGViewerControl";
   help_string: "PNG Image Control for DeLine's Thesis";
   IPNGViewerControl: player com_Interface with
      iid: UUID{7cf18aa2-36ff-11d2-9fd5-00104b33709d};
      help_string: "PNG Image Interface";
      FileName: player COM_Property with
         id: 1;
         signature: "BSTR";
      end
   end
end
```

A packaging description is a structured collection of parameters, where each parameter has a name and a value. Values, shown underlined above, are either simple datatypes like strings (the value of the *name* parameter) or integers (the value of the *id* parameter) or are the result of running a packaging-specific tool, like ActiveX's Genguid (the value of the *library_iid* parameter). These parameters capture high-level decisions about the particular packaging. For an ActiveX control, these include: machine-readable names for the control (*library_iid*, *co-*class_iid); a human-readable name of the control (*name*); documentation describing the control (*help_string*); and the COM interfaces that the control exports. The COM interfaces that are common to most ActiveX controls, like those for windowing and painting, are implicit in this packaging description; only those COM interfaces that are specific to this control are explicitly listed. The described ActiveX control exports one COM interface (*IPNGViewerControl*) in addition to these implicit common interfaces. The parameters to this COM interface include: a machine-readable name for the interface (*iid*); documentation describing the interface (*help_string*); and a description of the property that this COM interface exports (*FileName*). In turn, the description of this COM property includes two parameters: an identification number for the property (*id*); and type of the COM property's value (*signature*). In this way, a packaging description can be highly nested, structuring the necessary parameters into bundles convenient to the type of packaging being described. This packaging description is derived from a definition, named *ActiveXControlPackaging*, that describes the parameters that must be included (their names and types) and how the parameters are grouped into bundles.

The definition of ActiveXControlPackaging is given on page 106

Once our Visual Basic developer writes this packaging description, her next step is to run a tool called Packgen on it. From this packaging description, Packgen automatically produces the source code and construction and installation steps necessary to achieve the described packaging. The source code that Packgen creates is called the *packager*, and, like the ware, the packager is summarized by a channel signature. Packgen generates both the packager's source code and its channel signature. The channel signature of the described ActiveX control packager is the following:

```
Pack =   out(Begin) → Calls.
Calls =  out(SetFileName) → Calls
        [?] out(Paint) → Calls
        [?] out(Finalize) → DONE.
```

The current Ciao tools do not support channels of type void, i.e. dataless channel . Instead, I use integer channels with dummy values as an approximation. The channels Begin and Quit are examples of this convention.

```
channel out int Begin;
channel out BSTR SetFileName;
channel out struct {struct { long left, bottom, top, right; }* rect;
                    void* hdc;} Paint;
channel out int Quit;
```

This packaging description is similar to the previous one, but uses one new feature. The rule Calls states that the packager can report data to any one of three channels: SetFileName, Paint, and Finalize. In this case, the use of the bracketed question mark ([?]) indicates that it is up to the packager to determine which of these three behaviors occurs. Whereas a ware's channel signature represents the essential interaction commitments needed to support the ware's functionality, a packager's channel signature represents the essential interaction commitments engendered in the described packaging.

Our Visual Basic developer's final task is to show the correspondence between the ware's channel signature and the packager's, in order to link them together. To do this, she creates the following table, called a *channel map:*

Initialize	Begin	

NewFile	SetFileName	channel in BSTR windows_string;
		channel out char* c_string;
		BSTR bstr;
		char* cstr = (char*)malloc(100);
		in(windows_string, bstr);
		sprintf(cstr, \"%S\", bstr);
		out(c_string, cstr);

Finalize	Quit	

Each row of the channel map specifies the correspondence between a channel in the ware's signature and one in the packager's. If the two channels do not share a common data representation, the third column contains a fragment of code that converts between the two data representations. When the corresponding channels in the ware and the packager share the same name and data representation (for example, the channel Paint), the correspondence can be omitted from the channel map. In this channel map, Initialize/Begin and Finalize/Quit differ only in their names. NewFile and SetFileName differ both in their names and data representations. The channel NewFile communicates strings in the standard C representation (char*); the channel SetFileName, in a COM-specific string representation (BSTR). The code above translates between these two string representations.

Once she creates the channel map, our Visual Basic developer runs the Flexible Packaging system on the ware, the packager, and the channel map. The ware and the packager should be thought of as two halves of a final run-time behavior: the ware achieves the functionality; the packager, the interactions. Only when they are linked together do they form a whole component. Given the ware, the packager, and the channel map, the Flexible Packaging system first uses the ware's and packager's channel signatures to test whether they are compatible. Although an exact definition of compatibility appears in Chapter 3, an intuition for why the PNG ware and packager are compatible can be gained by inspecting their conditionals. The packager wants to choose which service is invoked, and the ware wants the "outside world" to choose which service is invoked. Their compatibility stems from an agreement about which module will drive the computation. If both wanted to drive the computation or if each wanted the other to drive the computation, they could not be sensibly combined. Once the Flexible Packaging system ensures compatibility between the ware and packager, it then uses packaging-specific construction and installation steps to produce the final component: an ActiveX control that parses and paints PNG images.

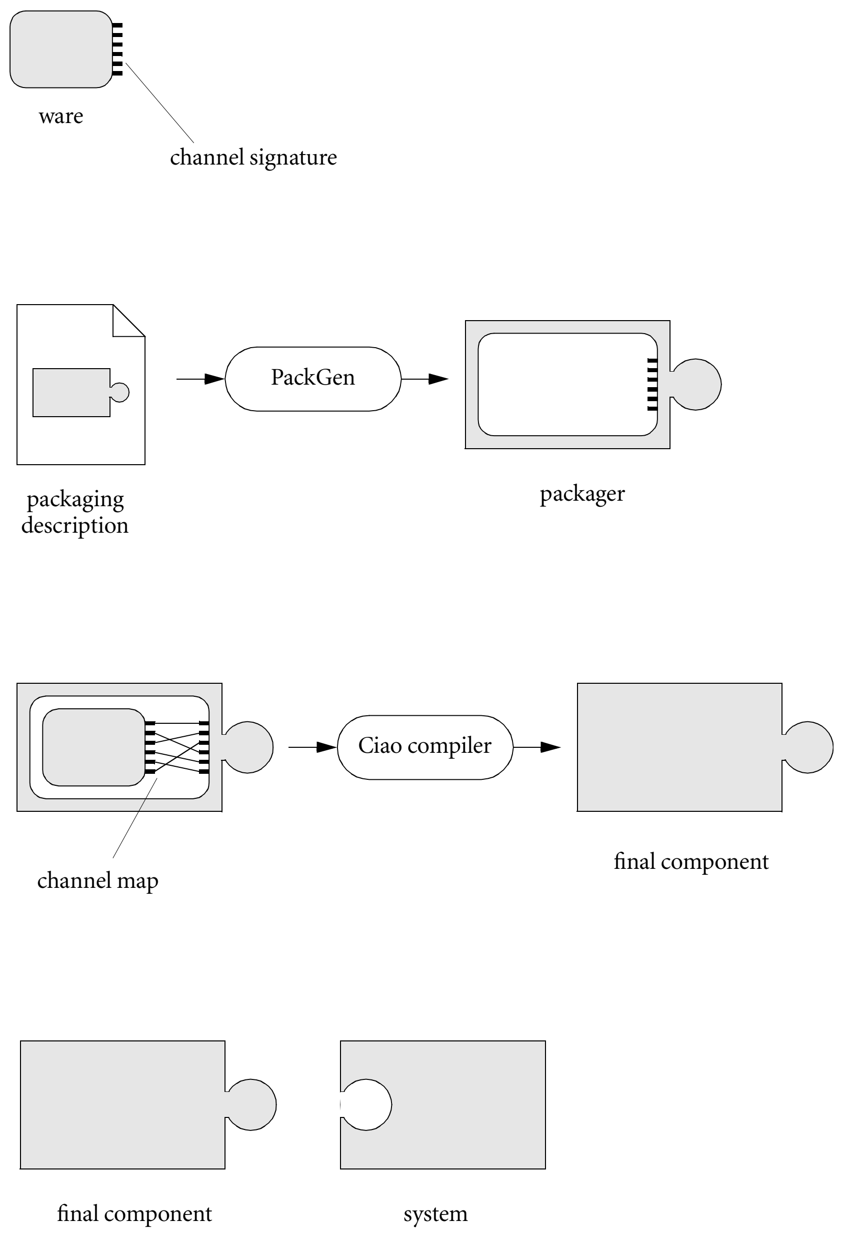
Figure 1.1 summarizes the process that our Visual Basic developer follow to create the PNG ActiveX control. Traditionally, a Visual Basic programmer who wants to reuse a PNG ActiveX control faces two choices: looking for the component in ActiveX repositories in the hope of finding one; or finding a PNG component with a different packaging and creating the necessary glue code to integrate it. Instead, with Flexible Packaging, she looks for a ware in component repositories (which are no longer packaging-specific niche markets) and then tailors the found ware to be packaged as an ActiveX control to suit her specific integration needs.

1.6 ORGANIZATION OF THE DISSERTATION

The remainder of the dissertation discusses the Flexible Packaging method in detail. Chapter 2 makes the notion of packaging more precise by decomposing it into a set of constituent aspects. This decomposition allows a discussion, in Chapter 6, of those aspects of packaging that Flexible Packaging does and does not address. Chapter 2 also catalogues the various techniques that practitioners have traditionally used to overcome packaging mismatch. Each of the eight techniques is described as a pattern (problem/solution pair). This use of a common framework

The Flexible Packaging method originally appeared in the 1999 International Conference on Software Engineering.[12]

FIGURE 1.1 The process by which the system architect tailors a ware's interactions to suit the integration context.



The architect selects a ware that implements her desired functionality. The ware is characterized by a channel signature.

The architect writes a packaging description that captures the packaging that she wants the final component to have. She runs PackGen on the description to produce the packager. The packager is also characterized by a channel signature.

The architect creates a correspondence between the ware's and packager's channels, called a channel map. She runs the Ciao compiler on the ware, packager, and channel map to produce the final component.

Because the architect tailors the component's packaging to the requirements of the integration context, the final component is ready for direct integration.

allows the techniques to be seen, not as an *ad hoc* collection, but as elements in a space of possible techniques. Flexible Packaging is shown as an element in this space.

Chapter 3 introduces the two major ideas underlying Flexible Packaging. First, in order to prevent the system architect from being overburdened with the details of a given type of packaging, the Flexible Packaging method introduces a new role into the system's development: the packaging specialist. The packaging specialist is responsible for knowing all about a given type of packaging (say, ActiveX controls) and for encapsulating his knowledge in the form of a software generator. Second, Chapter 3 discusses how the mix-and-match relationship between wares and packagers challenges today's modularity mechanisms and motivates the need for channels as a new modularity mechanism. With these main ideas established, Chapter 4 then describes the tools and skills that the ware provider, the packaging specialist, and the system architect each need to play his or her role in system development.

The last two chapters describe how I evaluated the Flexible Packaging method. Chapter 5 describes a series of experiments in which I combined three wares and nine packaging generators to produce thirteen different components. Chapter 6 analyses these experimental results: it describes how the packagings chosen for experimentation represent the complexities that hinder today's practitioners; it describes those aspects of packaging that the method addresses and those it does not; and it describes, property by property, how the experimental results validate the thesis claim. The chapter concludes with ideas for future research directions and a review of the major contributions of Flexible Packaging.

2 *Packaging & Packaging Mismatch*

For decades, system developers have faced and surmounted various forms of packaging mismatch. As mentioned in the previous chapter, a common solution is to interpose "glue code" (sometimes called wrappers, bridges, mediators, or adaptors) between the component being integrated and the rest of system to compensate for differences in packaging. Another typical solution, often used when the "glue code" solution is inapplicable, is to modify the component's source code to change its packaging in order to integrate it into a new system. Unfortunately, the collective experience about resolving packaging mismatch currently exists only as unrecorded folklore and as technique-specific technical papers scattered throughout the computer science literature. As witnessed by the number of different words for "glue code," we even lack a consistent, precise vocabulary to describe the problem and its solutions.

In order to allow system integrators to attack packaging mismatches systematically, what is known about the problem and its solutions should be assembled and organized, in much the same vein as the patterns community is organizing object-oriented design.[18] This chapter takes a step in that direction. First, I introduce a more precise vocabulary for describing a component's packaging by decomposing the notion of packaging into a set of identifiable aspects. I then present a catalog of packaging mismatch resolution techniques, with examples of each technique drawn from a variety of architectural styles and classified according to the new vocabulary. The emphasis of this catalog is not to present any given technique in enough detail to form a ready recipe for solving a given integration problem. Instead, the emphasis is to gather and organize a wide variety of techniques and to highlight their relationships and to allow ready comparison among them. The chapter concludes with a discussion of related classification efforts. In the next chapter, after I introduce a more complete picture of how Flexible Packaging works, I show how Flexible Packaging fits into the space of techniques that this catalog defines.

2.1 ASPECTS OF COMPONENT PACKAGING

A common way to describe a packaging mismatch problem is by reference to a difference in interaction mechanism: "I'd like to reuse this module written in C, but the components in my system interact by announcing events, not by making procedure calls." Describing the difference this way, however, gives no feel for how different the two packagings really are, for example, how different procedure call is from event announcement. In order to be more precise about the nature of a packaging mismatch, it is useful to decompose a component's packaging into a set of aspects. An instance of packaging mismatch can then be described as a difference in one or more of those aspects. This section provides such a set of aspects, each of which is described in a subsection below. This new vocabulary is used in Section 2.2 to describe examples of various techniques in the catalog and later in Chapter 6 to evaluate the Flexible Packaging method.

2.1.1 *Data representation*

In order for two components to transfer or share a data item without mismatch, they need to agree on its representation. For small-scale data items, like basic data types, this agreement commonly means either that (1) the components share a common type system and agree on the data item's type or that (2) the components agree on a bit-level representation of the data item, for example, the IEEE floating point standard representation. For large-scale data items, like files and databases, this agreement commonly means that the components agree on the data item's format or syntax. For these larger data items, whether the components agree about data representation is not necessarily a black and white issue. For example, one vendor's word processor may be capable of editing documents created with another vendor's word processor, but the document may lose some of its formatting when opened in the foreign word processor. Whether such a loss constitutes a data representation mismatch is context-dependent and hence up to the system integrator. Also, although the differences in data representation may reflect deeper semantic mismatches, semantic mismatch is outside the scope of packaging mismatch and is not discussed here. This distinction between semantic and packaging mismatch is useful since the former problem requires human understanding to solve, whereas the latter is amenable to automatic solutions.

2.1.2 *Data and control transfer*

Components interact by sending each other data or by transferring control between them (the thread of control leaves one component and enters another, as with remote procedure call). In order for two components to transfer data or control without mismatch, they must agree on the mechanism to use and the direction of the transfer. When two components do not agree on the mechanism by which to exchange data/control, we would like a way to judge how bad the mismatch is. To do this, we can consider three facets of the transfer: what is transferred, data or control; the direction of the transfer; and whether the transfer occurs at the sender's request (often called "push") or at the receiver's request (often called "pull"). Popular mechanisms by which a components pulls data include:

- environment variable;
- text stream;
- socket (stream style);
- file;
- shared variable;
- shared memory; and
- database access (through queries).

Popular mechanisms by which a component is pushed data and control include:

- procedure call;
- remote procedure call;
- event;
- message;
- socket (message style);
- property set; and
- database access (through update triggers).

Finally, popular mechanisms by which a component is pushed control include:

- exception; and
- interrupt.

This push/pull distinction can be used to judge the degree to which mechanisms differ. Overcoming a difference between mechanisms is easier for those mechanisms that agree about whether data/control is pushed or pulled. For example, getting an environment variable and fetching from shared memory are similar because they agree on direction (transfer in), on what is transferred (data), and on reception request (pull). However, reading from a data stream is different from being notified of an event. Although both mechanisms involve transferring data into the component, the former is done at the receiver's request (pull); the latter, at the sender's request (push).

2.1.3 *Transfer protocol*

In order for two components to interact without mismatch, they must agree on the overall protocol for transferring data and control. At minimum, this means agreeing on the number and order of individual transfers of data or control. For example, for message-based data exchanges, this may take the form of both components agreeing on a standard message-passing protocol. For procedure-based interaction, it may take the form of each procedure caller upholding the called procedure's preconditions. For communication styles where communication speed is a factor, like modem or satellite communication, the transfer protocol aspect may include timing considerations.

2.1.4 *State persistence*

A component may vary in the degree to which it retains state between interactions. As an example of state scope, consider two versions of the interactive fiction game, called Adventure. The original version of this game is played on a terminal. The game iteratively prints text that reports the state of the game, prompts for a move, and then uses the move to update the state of the game. This iteration repeats until the players wins, loses, or quits the game. Starting with the orginal version, Wu reimplemented the game to be played through the web.[52] With his version, a player reads a web page to learn the state of the world, enters text in a form, and clicks a button to submit his command to a CGI script. The CGI script uses the player's move to produce a web page with the new state of the game.

The chief difference between the game's terminal-based packaging and its CGI-based packaging is in state persistence. The terminal version runs continuously and retains the game state in memory. The CGI script is executed anew each time a player makes a move. For the CGI script to have access to the game state, the web form passes the CGI script both the player's move and an encoding of the current game state. In short, in terminal-based interaction, the state persists for an arbitrary number of moves; whereas, in CGI-based interaction, the state persists for only a single move.

2.1.5 *State scope*

A component may vary in the amount of its internal state it allows other components to affect. For example, a document editor with a programmable interface may allow interactions that affect the entire state of the editor component (e.g. a "quit application" operation), or a whole document ("save", "print"), or a portion of a document ("delete paragraph"). If a component interacts with several other components simultaneously (for example, a server that interacts with multiple clients), then it may divide its internal state into individual pieces of state for each component with which it interacts. When two components disagree over the amount of state to be affected during an interaction, this is an instance of state scope mismatch.

2.1.6 *Failure*

Components vary in the degree to which they tolerate interactions that fail. For example, a component that reads from a file is typically designed to expect the data from the file to be delivered reliably and accurately; whereas, a component that uses unreliable network message passing is typically written to tolerate missing or garbled data. Component also vary in the extent to which they themselves fail. This aspect captures whether individual transfers of data or control may fail. How these failures fit into the overall ordering of transfers is part of the Transfer Protocol aspect.

2.1.7 *Connection establishment*

A component's packaging consists not just of the details of the interaction mechanisms it uses but also in how those mechanisms are set up and torn down. Consider a component that is packaged to read a file. The architectural connection between the component and the file it reads could be established in a variety of ways: the component may open and close a file with a hard-coded name; the component may open and close a file whose name is given through interaction with a user; another component in the system may provide the name of a file or a file descriptor. For two components to interact without mismatch, they must agree on how the interaction mechanism they use is set up and torn down.

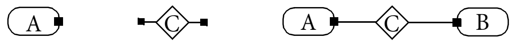
2.2 A CATALOG OF MISMATCH RESOLUTION TECHNIQUES

This catalog originally appeared in the 1999 Symposium on Software Reusability.[13]

Following the lead of the patterns community, this section provides a catalog of techniques for resolving packaging mismatch, where each technique is described in a template form. The template provides the following information: a short name for the technique; a schematic diagram that captures the gist of the technique; a more detailed prose explanation of the technique; and a set of examples of the technique in use. While many pattern languages are intended to be highly situated to allow specific guidance to be given, the patterns here are intentionally abstract to promote comparison among the techniques described.

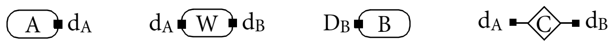
In the catalog's characterization, what principally distinguishes one technique from another are the set of packaging commitments that are made, the binding time when the commitments are made, and the architectural elements that embody the commitments. To illustrate the notion of a packaging commitment, consider a developer who, when implementing a module, chooses to report errors that occur in the module's functions by setting a global integer variable called *errno* and by returning zero from the function. He has made several commitments about interaction, including: a data representation commitment (the variable is an integer); a data transfer commitment (shared variable is the mechanism); a transfer protocol commitment (the variable can be read after every call to a function that returns zero); and a state scope commitment (the variable is global). We would say that these commitments are made when the module is developed and that they are embodied in the module.

The template uses a schematic diagram to show at a glance how the system architecture is transformed to resolve the packaging mismatch and to allow ready comparison of the techniques. Each diagram is a series of rows representing significant times during the development of the system, when either commitments are made or the architecture changes. Borrowing from architectural description languages (ADLS)[48], a system is described as a configuration of components and connectors. Connectors, like pipes, procedure calls, and message passing, mediate the interaction among components. In the diagrams, components are depicted as labeled, round-cornered boxes; connectors, as labeled diamonds. Because a component may interact in multiple ways (for example, by both reading a file and sending a message), the diagrams use a black square, called a port, to depict each of the ways a component interacts. Similarly, a connector provides multiple roles for the components' ports to play, which are also depicted with black squares. For example, a pipe has a data source role and a data sink role. Here is a picture of an isolated component, an isolated connector, and a configuration of two components interacting through a connector:

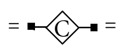


For simplicity and uniformity, these diagrams depict only binary connectors, although the diagrams could be generalized to *n*-ary connectors by depicting the connectors with more arms. Whereas ADLS typically use different shapes to discriminate among different types of components and connectors, this notation intentionally uses the same shape regardless of type because the strategies that follow are applicable to more than one type of component and connector. Instead, these diagrams use annotations to highlight the commitments that the components and connectors make about interaction.

To show the commitments that are embodied in these architectural elements, the pictures of ports and roles are annotated with labels. A lowercase *d*, often subscripted, is used to denote a particular decision about interaction; a capital *D* is used to denote a set of alternatives for a decision. Here are some examples of these annotated elements:



The component *A* above interacts through one port and is committed to some decision d_A on that port. For example, if *A*'s port represents the reading of a shared variable, then d_A might stand for a commitment about the data representation of that variable. Component *W* above interacts through two different ports, where a different commitment has been made about interaction through each of the ports. Using the same label on multiple ports, as with *A*'s and *W*'s use of d_A, means that the components agree about this commitment. Component *B*'s decision about interaction is limited to some set of alternatives D_B, but *B* is free to choose any one of those alternatives for its final commitment. For example, if component *B* were a word-processing application with a port for reading document files, the set D_B could stand for the set of document formats that the application is capable of reading, like MacWrite versus WordPerfect versus RTF. Finally, connector *C* has made a different commitment about interaction through each of its roles. It is very common for a connector to be indifferent about a commitment, so long as the same commitment is made for each of its roles. These commitmentinvariant connectors are depicted with equal signs at the roles:



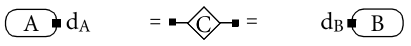
For example, a procedure call connector is indifferent about the number, order, and types of arguments passed from the procedure caller to the procedure definer, so long as the commitment is the same for the caller and definer.

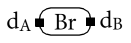
I organize these commitment diagrams into a series of steps, which represent how the system architect transforms the architecture in order to resolve the packaging mismatch. The steps in the transformation are given generic labels (s1, s2, s3) to convey the order of the steps while abstracting away from the time at which the step is performed. For instance, for a given technique in the catalog, step s2 may correspond to system integration time in one of its examples; in another example of the same technique, s2 may correspond to run time.

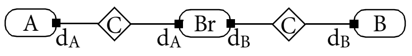
The examples that appear in the templates were chosen, not because they are the best or most representative examples of the technique, but because sufficiently detailed documentation about them is available. Because much of this documentation consists of research papers, there is a bias toward automated solutions. This in turn means that many of the examples are about data representation mismatch, a problem that lends itself more readily to automated solutions.

On-line Bridge

Schematic

(s1) 

(s2) 

(s3) 

Problem
To integrate components *A* and *B* that have commitments made at the time of their development that conflict with one another (s1). Connector *C* cannot arbitrate the difference in those commitments.

Solution
Introduce a new component *Br* that is capable of interacting in two ways: one way that is compatible with *A*'s commitment d_A; one that is compatible with *B*'s commitment d_B (s2). Interpose this component between *A* and *B* (s3). The component *Br*'s computation makes up for the differences between the commitments d_A and d_B. Although the examples below involve bridges that a tool generates, bridges are quite often developed by hand to suit the details of a particular mismatch. The bridge here is "on-line" in that it is part of the system's final control structure.

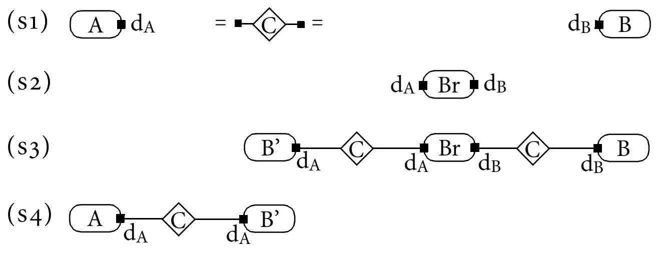
Variation
Component *A* is developed to interact through connector C_1; component *B*, through connector C_2. The introduced bridge *Br* interacts with *A* through C_1 and with *B* through C_2.

Examples

- Nimble[41]
  Aspect of packaging: data representation, namely the number, order, and types of arguments and results passed between a procedure caller and definer
  Component A: a procedure caller
  Component B: a procedure definer
  Connector C: procedure call
  Component Br: a Nimble-generated bridge, which accepts the parameters that *A* passes, calls B with the parameters *B* expects, accepts the result from *B*, and returns the result that *A* expects
- Yellin and Strom adaptor[53]
  Aspect of packaging: data representation and transfer protocol, namely the number and order of method calls between an object and a client of that object
  Component A: an object calling another object's methods
  Component B: the object whose methods are being called
  Connector C: method call
  Component Br: a generated "adaptor" (bridge) object, which accepts method calls from *A* and makes method calls on *B*

Off-line Bridge

Schematic



Problem
Same as for the On-line Bridge technique, with the restriction that component *B* is some form of persistent data (s1). The mismatch between *A* and *B* is about data representation.

Solution
Introduce a new component *Br* that is capable of reading data with representation d_B and writing data with representation d_A (s2). Component *Br* is typically a stand-alone tool. Run component *Br* to transform *B* into a new component *B*' (s3). Integrate *B*' with *A* (s4). Unlike the On-line Bridge technique, where the bridge is typically developed to suit the needs of a particular system, off-line bridges are often available as separate tools and can hence be acquired rather than developed. When d_A and d_B are about an aspect other than data representation, use the On-line Bridge technique so that the bridge can be part of the control structure of the final system. To automate the step of executing the off-line bridge and/or to select the bridge at run time, use the Mediator technique.

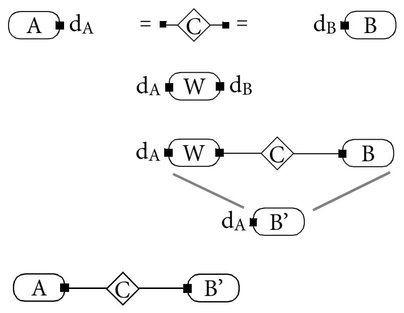
Examples

- Debabelizer
  Aspect of packaging: data representation, namely image format
  Component A: MacPaint, committed to MacPaint format
  Component B: an image in Photoshop format
  Connector C: file access
  Component Br: the tool Debabelizer, which can convert among many image formats, including MacPaint and Photoshop
- Word for Word
  Aspect of packaging: data representation, namely document format
  Component A: Microsoft Word application, committed to Word format
  Component B: a document in FrameMaker format
  Connector C: file access
  Component Br: the tool Word for Word, capable of converting among a variety of document formats, including Word and FrameMaker

Wrapper

Schematic

(s1)
(s2)
(s3)
(s4)
(s5)



Problem
Same as for the On-line Bridge Technique (s1).

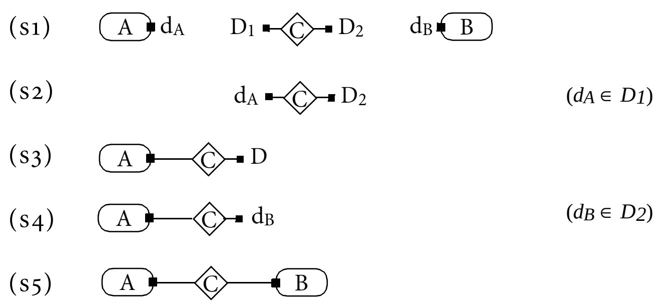
Solution
The solution is the same as with the On-line Bridge technique, with one additional step. Before the final integration, encapsulate the wrapper *W* (the analogue of bridge *Br*), the component *B*, and the connector between them within a new component *B*' (s4). This encapsulation step, depicted with the striped lines above, is about both abstraction and access: the component *B*' hides the commitment d_B inside its implementation; and component *B* can only be accessed through component *W*. Both these aspects of the encapsulation step simplify reasoning about the final integrated system. (Note that whether *A* or *B* is encapsulated with *W* is arbitrary in this chapter's formulation. In an actual system, system-specific considerations determine this choice. Typically, if *B* is encapsulated with *W*, it is because d_B represents a "legacy" commitment to be denigrated in favor of d_A in the system's future life.)

Examples

- Hardware emulator
  Aspect of packaging: data representation, namely instruction set. This data representation difference reflects significant semantics differences (e.g. RISC vs. CISC, different memory models), which must also be addressed but are not packaging mismatch problems.
  Component A: Intel x86 executable, committed to x86 instruction set
  Component B: Sun Sparc processor, committed to Sparc instruction set
  Connector C: Instruction fetch and execution
  Component W: program that runs on a Sparc and emulates the x86 processor
- Database wrapper[38]
  Aspects of packaging: data transfer and transfer protocol
  Component A: database accessing program, committed to SQL query language
  Component B: file formatted with newline-separated records, committed to linear access (no query language)
  Connector C: data access
  Component W: automatically generated component that accepts an SQL query and performs linear access to fulfil the query
- MacLink
  Aspect of packaging: data representation, namely floppy disk format (DOS vs. Macintosh)
  Component A: Macintosh application, committed to Mac formatted files
  Component B: a file on a DOS-formatted disk
  Connector C: file access
  Component W: MacLink, which makes a file on a DOS-formatted floppy disk appear to be a Mac-formatted file

Mediator

Schematic



Problem
To integrate components *A* and *B* that have commitments made at the time of their development that conflict with one another (s1). Connector *C* is simultaneously capable of supporting several alternatives for a given commitment, often about data representation. It does this by having an internal infrastructure that is able to choose and coordinate among specialized components, called brokers or agents, that are capable of handling a particular data translation. The infrastructure is often designed to allow the set of alternative commitments to be easily grown, even at run-time with some mediators. Mediator technology is currently an object of research and should become more available to system integrators over time.

Solution
Specialize the mediator so that its commitment is compatible with component *A*'s (s2); then integrate it with component *A* (s3). Repeat for component *B* (s4, s5). These two specialization and integration steps may take place at different times, for example, one at system integration time, one at run-time.

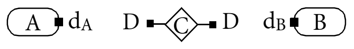
Variation
Connector *C* allows variation on only one of its roles, making a fixed commitment for the other role. For example, a mediator may support a fixed type of interaction with a data-consuming component, but be capable of interacting with many types of data-producing components.

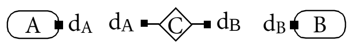
Examples

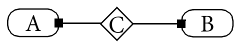
- Tom[37]
  Aspect of packaging: data representation, namely document format
  Component A: a program that reads PostScript documents
  Component B: a document in LaTeX format
  Connector C: the Tom service, which can convert among a variety of document formats by choosing the appropriate conversion tools
- Retsina[51]
  Aspect of packaging: data representation, namely the formats of different information sources in the same domain, like stock information
  Component A: a stock portfolio management program
  Component B: a web page showing periodic stock updates
  Connector C: the Warren system (an instance of the Retsina framework)

Intermediate Representation

Schematic

(s1) 

(s2)  $(d_A \in D, d_B \in D)$

(s3) 

Problem
Same as for the Mediator technique, with the restriction that the mismatch between *A* and *B* is about data representation. Connector *C* is simultaneously capable of supporting several alternatives for a given commitment about data representation. It does this by committing to its own choice for this alternative (call it d_I) and by implementing all translations to and from each of the alternatives in *D* and d_I. The advantage of having an intermediate form is that the number of translations the connector must implement grows linearly with the number of alternatives; whereas, the number of pairwise translations grows quadratically with the number of alternatives. The disadvantages are that the cost of two translations must be incurred even when *A* and *B* commit to the same alternative (d_{AB} to d_I to d_{AB}) and that the translations may lose information. Typically, the set of alternatives is committed when the connector is developed.

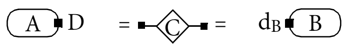
Solution
Specialize the connector to the mismatch at hand (s2) and integrate it (s3). Because the set of alternatives is typically fixed when the connector is developed, the system integrator often uses connector-specific tools at system build time to achieve the specialization and integration.

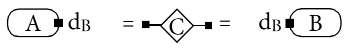
Examples

- Xerox PARC's Inter-Language Unification (ILU)[28]
  Aspect of packaging: data representation, namely representation of basic datatypes (integers, strings, booleans, records, etc.)
  Component A: a program written in C
  Component B: a program written in Lisp
  Connector C: the ILU infrastructure, which supports inter-language procedure call
- Corba
  Aspect of packaging: data representation, namely representation of basic datatypes (integers, strings, booleans, records, etc.)
  Component A: an object implemented in C++
  Component B: an object implemented in Smalltalk
  Connector C: a Corba ORB, which supports inter-language method call

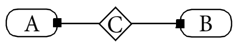
Unilateral Negotiation

Schematic

(s1) 

(s2) 

$(d_B \in D)$

(s3) 

Problem
To integrate components *A* and *B*, where component *A* is committed at its development time to a set of alternative decisions about interaction and component B is committed to a particular decision (s1).

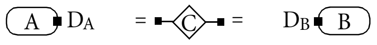
Solution
If component *B*'s commitment is in the set of commitments that component *A* is capable of supporting, then specialize component *A* to match *B*'s commitment (s2) and integrate the two (s3). This technique can also be seen as a mismatch prevention technique: develop components that support more than one style of interaction to make them more widely reusable. (One way to realize this advice is the Component Extension Technique.) If component *B*'s commitment is not in the set of commitments that component *A* is capable of supporting, then consider another technique, like On-line Bridge or Wrapper.

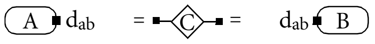
Examples

- Microsoft's COM connector[6]
  Aspect of packaging: transfer protocol, namely the interface (collection of procedures) by which A will export computation to B
  Component A: a COM component exporting multiple interfaces, for example, multiple versions of the same logical interface
  Component B: a COM component importing a particular interface, for example, a particular version
  Connector C: the COM connector
- "Fat" executables
  Aspect of packaging: data representation, namely processor instruction set
  Component A: a Macintosh "fat" executable, i.e. a program compiled to both the 68000 and PowerPC instruction sets, but provided as a single executable file
  Component B: a PowerPC processor
  Connector C: the MacOS program loader
- Optional procedure arguments
  Aspect of packaging: data representation, namely the number and types of arguments passed between a procedure caller and definer
  Component A: a procedure definer
  Component B: a procedure caller
  Connector C: a procedure call connector that allows for optional (keyword) arguments, as with Modula 3 or Common Lisp
- Views of relational databases
  Aspect of packaging: data representation, namely the grouping of data items into a record
  Component A: a relational database
  Component B: a database accessor
  Connector C: a DBMS, which allows a dynamic grouping of data (view) to be formed from the database's tables

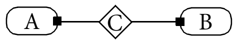
Bilateral Negotiation

Schematic

(s1) 

(s2) 

$(d_{ab} \in D_A \cap D_B)$

(s3) 

Problem
To integrate components *A* and *B*, each of which is committed at its development time to a set of alternative decisions about interaction (s1) and to a protocol for selecting one of the alternatives by negotiating with its partner components (s2). The negotiation may be either symmetric (the two components interact through a pre-determined channel to choose the alternative) or asymmetric (one component alone chooses the alternative).

Solution
Develop components that support negotiation to prevent packaging mismatch. There are currently too few examples of bilateral negotiation to provide general advice.

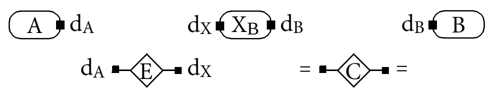
Examples

- Microsoft's COM connector
  Aspect of packaging: transfer protocol, namely the interface (collection of procedures) by which *A* will export computation to *B*
  Component A: a COM component exporting multiple interfaces, for example, multiple versions of the same logical interface
  Component B: a COM component capable of importing several interfaces. This component alone chooses the final interface by iteratively querying for the interfaces *A* supports. For example, it might seek the most modern version of an interface that *A* and *B* share in common.
  Connector C: the COM connector
- Modem whistling
  Aspect of packaging: transfer protocol, namely two communication parameters: modulation standard (bits per baud) and transmission rate (bits per second). There is a standard algorithm by which the two modems interact symmetrically to select the best values for these parameters.
  Component A: a modem making a call
  Component B: a model receiving a call
  Connector C: a bit stream channel (telephone line)

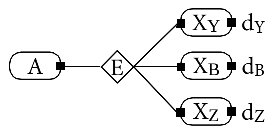
Component Extension

Schematic

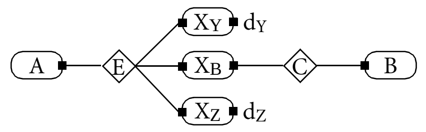
(s1)



(s2)



(s3)



Problem
To integrate an extensible component *A* to a component *B* with a fixed commitment about interaction. The developers of component *A* defer some commitments about interaction by delegating these commitments to a set of modules integrated when the component is initialized at runtime. When component *A* is developed, its designers commit to an interface between *A* and the dynamically loaded modules, called extensions, plug-ins, or add-ins and denoted above by *X* (s1). Either before or at the beginning of run-time, *A* is integrated with the extensions (s2), after which the set of alternative commitments that component *A* is capable of making is the union of those alternatives that the extensions committed to individually when they were developed. Later, when component *B* is integrated, component *A* selects the extension whose commitment agrees with d_B and integrates the extension and *B* (s3).

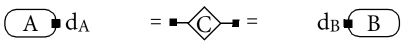
Solution
Develop an extension that matches component *B*'s commitment, and integrate it with component *A*. When seen from the point of view of component A's developers, this is a mismatch prevention technique; from the point of view of someone selecting or developing an extension, this is a repair technique. This technique provides a particular architecture for realizing Unilateral Negotiation. Namely, if component *A* and its extensions were encapsulated into a single component whose port had commitment $D_A = \{d_Y, d_B, d_Z\}$, then the diagram above would fit the pattern for Unilateral Negotiation.

Examples

- Word add-ins
  Aspect of packaging: data representation, namely document format
  Component A: Microsoft Word application
  Extensions X: Word add-ins, which can each read documents in different formats
  Component B: a FrameMaker document
  Connector C: file access
- Netscape plug-ins
  Aspect of packaging: data representation, namely document format, where a document is considered any information source whose contents can be displayed on a workstation (text, image, animation, sound, etc.)
  Component A: the Netscape web browser
  Extensions X: Netscape plug-ins, one per type of document, where the document's type is manifest through its file extension or MIME type declaration
  Component B: a document on the web
  Connector C: web document access (e.g. HTTP)

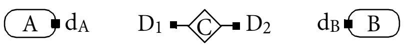
The techniques in this catalog form a space whose primary discriminator is the degree of flexibility engendered in the initial set of architectural elements to be integrated. There are three starting points.

2.3.1 *No exibility*



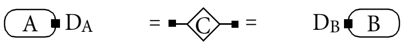
Because the components make particular and conflicting commitments about which the connector requires agreement, the integrator adds new components to the system to overcome the mismatch. The chief technique in this family is On-line Bridge, which has two specializations: Off-line bridge, where the mismatch is about data representation; and Wrapper, which involves an extra encapsulation step.

2.3.2 *Flexible connector*



The connector is designed to support a variety of commitments. When the components' commitments fall within the connector envelope of possibilities, the integrator or the connector itself specializes the connector to overcome the particular mismatch. When the components' commitments do not fall within the envelope, a technique from the "no flexibility" family may be used. The chief technique in this family is Mediator, which has one specialization: Intermediate Representation, where the mismatch is about data representation.

2.3.3 *Flexible component(s)*



One or both of the components are designed to support a variety of commitments about which the connector requires agreement. When there is overlap in the components' envelopes of possibilities, the integrator or the components themselves specialize the components to achieve agreement. The two main techniques in this family are Unilateral and Bilateral Negotiation. (Unilateral Negotiation is not truly a specialization of Bilateral Negotiation, since the latter can involve symmetric solutions to mismatch resolution, like modem whistling, which the former cannot.) Unilateral Negotiation has one specialization, Component Extension, which provides a particular architecture for achieving the more general technique.

2.4 FLEXIBLE PACKAGING IN THIS SPACE OF TECHNIQUES

The Flexible Packaging method is an example of Component Extension. With reference to the catalog entry on page 31, component *A* is the ware, component *X* is the packager, and connector *E* is the use of channels. The examples in the catalog, Word and Netscape plug-ins, load their extensions when the application is initialized and select the proper extension when the connection to component *B* is made at run-time. In contrast, with Flexible Packaging, only one extension can exist at run-time, and the system integrator selects and integrates this extension at system integration time.

2.5 RELATED CATALOGS AND CLASSIFICATIONS

The catalog in this chapter builds on the work of the software architecture community, which has argued for making the types of interaction among components first-class abstractions, called connectors.[48] The schematics in the catalog directly reflect this style of system decomposition. Within this community, Shaw has argued that extra-functional properties of software components, like packaging, often play as important a role during system integration as functional properties.[45] She provided a preliminary list of packaging mismatch resolution techniques and called for it to be "elaborated and refined." The catalog in this chapter is in response to that call.

In a similar vein to this cataloguing work, other researchers have classified various aspects of software architecture to begin to bring discipline to today's folklore. Shaw and Clements classified the architectural styles that a system may have, based in part of the interaction mechanisms used in the system.[46] In a companion paper, Kazman, Clements, Bass, and Abowd classified software components and the interactions among them (connectors) both by how they compose to form systems and by how they behave at run time.[30] Like this chapter, their model of runtime behavior is based on the transfer of data and control among components, though this chapter views this behavior in more detail.

Garlan, Allen, and Ockerbloom report the problems that they experienced in building a system from reusable parts.[19] They label these problems under the general heading of *architectural mismatch* and provide an initial decomposition of architectual mismatch into four categories: assumptions about components; assumptions about connectors; assumptions about the global architecture; and assumptions about the construction process. The concept of architectural mismatch is more general than that of packaging mismatch and includes non-packaging problems like whether components are built from redundant infrastructures and whether different components' construction processes mutually interfere. Also, their categorization of architectural mismatch does not perfectly align with the concept of packaging mismatch, since their categories of problems about components and problems about connectors both include packaging mismatch problems.

As an illustration of how patterns can capture expertise, Mularz reports four patterns for solving integration problems.[35] Mularz's patterns are more specific and situated than those presented here. For example, her Wrapper pattern, akin to the pattern of the same name here, deals with components that expose an API that is no longer to be used. Whereas her patterns are meant to provide specific guidance about particular integration problems, the patterns here are more generic in order to promote ready comparison among them.

Similarly, Dellarocas created an initial handbook of system integration problems paired with solutions, more comprehensive in scope than Mularz's.[16] Dellarocas' classification of interaction problems includes more problems than just packaging mismatch (for example, timing dependencies between components) but is less detailed in the regions of overlap with this chapter.

3 *Separating Functionality from Packaging*

The basic idea behind the Flexible Packaging method is to take the decisions about a software component's packaging out of the hands of the component provider and put them into the hands of the system architect. This chapter describes the two key elements of the design of the Flexible Packaging method that achieve this shift in responsibility.

First, Section 3.1 describes the need for a new participant in the development and deployment of software components. A given type of component packaging typically involves a lot of specialized skills and knowledge. By shifting the responsibility for the component's packaging from the component provider to the system architect, we relieve the component provider from knowing these packaging details. However, we do not want this shift to cause the system architect to become burdened with the packaging details. To shelter the system architect from these details, the Flexible Packaging method introduces a new development role: the packaging specialist. With this introduction, the Flexible Packaging method distributes the responsibility for a component among three participants: the ware provider captures the component's functionality in a module called the ware; the system architect makes the high-level decisions about a component's packaging; and the packaging specialist uses the architect's high-level decisions to produce the detailed code and artifacts needed to achieve the packaging that the architect describes. This section describes both the nature of the packaging details that the packaging specialist hides and how he hides them.

Second, Section 3.2 describes how the Flexible Packaging method requires the ability to combine modules that are independently developed – one module that implements a component's functionality, one module that implements its interactions. If the modules' interfaces are based on procedure calls (or its variants), the module integrator often must create "glue code" to integrate the modules – a situation that Flexible Packaging seeks to avoid. Instead, with Flexible Packaging, the modules run concurrently and interact through data channels. This approach allows independently authored modules to be mixed and matched, with any needed glue code semiautomatically generated.

After the introduction of these two key elements in the design of the Flexible Packaging method, Section 3.3 discusses the previous work that is most directly related to Flexible Packaging. Section 3.4 then reviews each of the three roles in the method to lay the groundwork for Chapter 4, which describes each role in detail.

3.1.1 *The complexities of component packaging*

For a good introduction to COM and ActiveX, consider Chappell.[10] The book's main content is blessedly light on jargon and technical arcana, and the running executive summary in the margin provides a particularly quick way to learn the basics.

Today, for a component provider to ensure that his component has a particular packaging, he must often exercise specialized skills and knowledge. As mentioned in Chapter 1, his choice of packaging typically has implications both on the content of the component's source code and on the tools and steps needed to construct and install the component. To appreciate this point in more detail, this section describes what it takes to package a component as an ActiveX control, which is representative of the skills and knowledge needed for many packagings. Although there are many ways to produce ActiveX controls, this explanation will focus on one representative approach, namely the use of Microsoft's Active Template Library (ATL).[21]

An ActiveX control is a special type of dynamic library that exports a collection of COM interfaces. Each COM interface exports a collection of procedures, called COM methods, and data items, called COM properties. COM itself is a run-time mechanism that implements the method calls and the property accesses. To hide many of the details of the COM mechanism, Microsoft created ATL, a C++ template library for implementing controls. Using this library, an ActiveX control is implemented as a subclass of one or more library classes. An example class that implements an ActiveX control is shown in Figure 3.1. The number of class and macro definitions used in this example illustrates the amount of packaging-specific knowledge a component provider must master to write his code, even when he uses a toolkit that simplifies the task.

Furthermore, using the ATL toolkit when writing the source code is not enough to package a component as an ActiveX control. The component provider must also carry out ActiveX-specific construction and installation steps. First, he must create a file written in Microsoft's Interface Definition Language (IDL), which describes the COM interfaces that the control exports. This description both serves as documentation for the control and is needed to link the control with other controls. Figure 3.2 shows an example IDL description. After this description is written, the component provider compiles it with an IDL compiler. Next, the component provider produces a registry file, which describes how the control should be entered into the Windows system registry. An example registry file is shown in Figure 3.3. To associate the compiled IDL file, the registry file, and an optional icon with the library implementing the control, the component provider creates a resource file for the library. An example resource file consists of the following lines:

```
101 REGISTRY "PNGViewerControl.rgs"
1 TYPELIB  "PNGViewerControl.tlb"
3 ICON "PNGViewerControl.ico"
```

```
class ATL_NO_VTABLE CPNGControl :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CPNGControl, &CLSID_PNGControl>,
    public CComControl<CPNGControl>,
    public IDispatchImpl<IPNGControl, &IID_IPNGControl, &LIBID_PNGControlLib>,
    public IProvideClassInfo2Impl<&CLSID_PNGControl, NULL, &LIBID_PNGControlLib>,
    public IPersistStreamInitImpl<CPNGControl>,
    public IPersistStorageImpl<CPNGControl>,
    public IQuickActivateImpl<CPNGControl>,
    public IOleControlImpl<CPNGControl>,
    public IOleObjectImpl<CPNGControl>,
    public IOleInPlaceActiveObjectImpl<CPNGControl>,
    public IViewObjectExImpl<CPNGControl>,
    public IOleInPlaceObjectWindowlessImpl<CPNGControl>,
    public IDataObjectImpl<CPNGControl>,
    public ISpecifyPropertyPagesImpl<CPNGControl>
{
public:
    CPNGControl();
    ~CPNGControl();

DECLARE_REGISTRY_RESOURCEID(IDR_GENERICCONTROL)

BEGIN_COM_MAP(CPNGControl)
    COM_INTERFACE_ENTRY(IPNGControl)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY_IMPL(IViewObjectEx)
    COM_INTERFACE_ENTRY_IMPL_IID(IID_IViewObject2, IViewObjectEx)
    COM_INTERFACE_ENTRY_IMPL_IID(IID_IViewObject, IViewObjectEx)
    COM_INTERFACE_ENTRY_IMPL(IOleInPlaceObjectWindowless)
    COM_INTERFACE_ENTRY_IMPL_IID(IID_IOleInPlaceObject, IOleInPlaceObjectWindowless)
    COM_INTERFACE_ENTRY_IMPL_IID(IID_IOleWindow, IOleInPlaceObjectWindowless)
    COM_INTERFACE_ENTRY_IMPL(IOleInPlaceActiveObject)
    COM_INTERFACE_ENTRY_IMPL(IOleControl)
    COM_INTERFACE_ENTRY_IMPL(IOleObject)
    COM_INTERFACE_ENTRY_IMPL(IQuickActivate)
    COM_INTERFACE_ENTRY_IMPL(IPersistStorage)
    COM_INTERFACE_ENTRY_IMPL(IPersistStreamInit)
    COM_INTERFACE_ENTRY_IMPL(ISpecifyPropertyPages)
    COM_INTERFACE_ENTRY_IMPL(IDataObject)
    COM_INTERFACE_ENTRY(IProvideClassInfo)
    COM_INTERFACE_ENTRY(IProvideClassInfo2)
END_COM_MAP()

BEGIN_PROPERTY_MAP(CPNGControl)
    PROP_PAGE(CLSID_StockColorPage)
END_PROPERTY_MAP()

BEGIN_MSG_MAP(CPNGControl)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_SETFOCUS, OnSetFocus)
    MESSAGE_HANDLER(WM_KILLFOCUS, OnKillFocus)
END_MSG_MAP()

public:
    STDMETHOD(get_FileName)(/*[out, retval]*/ BSTR *pVal);
    STDMETHOD(put_FileName)(/*[in]*/ BSTR newVal);
private:
    BSTR cached_FileName;
    HRESULT OnDraw(ATL_DRAWINFO& di);
};
```

FIGURE 3.1 An example C++ class declaration for an ActiveX control written using Microsoft's Active Template Library.

FIGURE 3.2 An example interface description in Microsoft's Interface Definition Language.

```
#include <olectl.h>
import "oaidl.idl";
import "ocidl.idl";

[
    object,
    uuid(7CF18AA2-36FF-11d2-9FD5-00104B33709D),
    dual,
    helpstring("PNG Image Interface"),
    pointer_default(unique)
]
interface IPNGViewerControl : IDispatch
{
    [propget, id(1), helpstring("")] HRESULT FileName([out, retval] BSTR *pVal);
    [propput, id(1), helpstring("")] HRESULT FileName([in] BSTR newVal);
};


[
    uuid(7CF18AA0-36FF-11d2-9FD5-00104B33709D),
    version(1.0),
    helpstring("PNG Image Control for DeLine's Thesis")
]
library PNGViewerControlLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [
        uuid(7CF18AA1-36FF-11d2-9FD5-00104B33709D),
        helpstring("PNGViewerControl Class")
    ]
    coclass PNGViewerControl
    {
        [default] interface IPNGViewerControl;
    };
};
```

This resource file is compiled with a resource compiler, and the compiled resource file is presented to the linker along with the compiled source code. In short, in addition to using ATL to produce his source code, the component provider must produce three non-code artifacts, each in its own syntax, invoke a packaging-specific tool, and use a packaging-specific switch to the linker.

To be fair, Microsoft's development environment, Developer Studio, is capable of hiding some of these steps and reducing the tedium of others. However, the skills and knowledge illustrated with this ActiveX example are in fact representative of the complexity that a component provider often faces when working with a given component packaging. Notice

```
HKCR
{
    PNGViewerControl.PNGViewerControl.1 = s 'PNGViewerControl Class'
    {
        CLSID = s '{7CF18AA1-36FF-11d2-9FD5-00104B33709D}'
    }
    PNGViewerControl.PNGViewerControl = s 'PNGViewerControl Class'
    {
        CurVer = s 'PNGViewerControl.PNGViewerControl.1'
    }
    NoRemove CLSID
    {
        ForceRemove {7CF18AA1-36FF-11d2-9FD5-00104B33709D} =
                                                        s 'PNGViewerControl Class'
        {
            ProgID = s 'PNGViewerControl.PNGViewerControl.1'
            VersionIndependentProgID = s 'PNGViewerControl.PNGViewerControl'
            ForceRemove 'Programmable'
            InprocServer32 = s '%MODULE%'
            {
                val ThreadingModel = s 'Apartment'
            }
            ForceRemove 'Control'
            ForceRemove 'Programmable'
            ForceRemove 'Insertable'
            ForceRemove 'ToolboxBitmap32' = s '%MODULE%, 1'
            'MiscStatus' = s '0'
            {
              '1' = s '131473'
            }
            'TypeLib' = s '{7CF18AA0-36FF-11d2-9FD5-00104B33709D}'
            'Version' = s '1.0'
        }
    }
}
```

FIGURE 3.3 An example registry file for an ActiveX control.

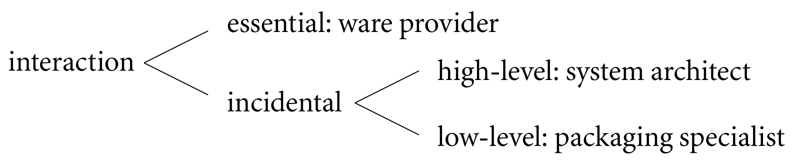
that the skills and knowledge needed to achieve a given component packaging are distinct from those needed to implement a given piece of functionality, like painting PNG images, calendar management, vehicle navigation, and so on. Hence, shifting the packaging decisions away from the component provider 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 architect, whose job is to understand the system-level implications of choosing different interaction mechanisms. Hence, in shifting packaging decisions to the architect, we do not also want to shift responsibility for the details that follow from the packaging decisions.

3.1.2 *The packaging specialist*

Although the Flexible Packaging method seeks to let the system architect choose a component's packaging, exposing her to the full complexity of a packaging's rules and rituals would make her job considerably harder than it is today. In current practice, a system architect must understand how to compose components packaged in a given way, but need not understand how to achieve that packaging. For example, many Visual Basic developers understand how to use ActiveX controls in their systems, without understanding how ActiveX controls are built. This exposure to packaging complexity would be especially burdensome to those architects who build their systems in heterogeneous architectural styles, which involve several types of packaging at once.

Further, much of this packaging complexity is uninteresting: some packaging decisions are high-level and represent an interesting part of a component's design; the other packaging decisions are mere details. For instance, the high-level choices for ActiveX controls include the choice of which COM interfaces a given control exports and the choice of the methods and properties that constitute a given COM interface. How these high-level choices are encoded in artifacts like IDL files is a detail of the technology. To prevent the system architect from being burdened with these low-level details, the Flexible Packaging method introduces a new participant into the process of developing and deploying components: the packaging specialist. A packaging specialist is expert in a given packaging technology, like ActiveX, and encapsulates the arcane rites necessary to ensure that a component has that packaging.

Reviewing from Chapter 1, the Flexible Packaging method distinguishes the essential decisions about a component's interactions from the incidental decisions. The essential decisions about interaction are intrinsic to the component's functionality and are hence the responsibility of the ware provider. The remaining incidental decisions about the component's interactions constitute the component's packaging. These packaging decisions themselves can be split into high-level and low-level decisions. The system architect makes the high-level packaging decisions; the packaging specialist, the low-level. In short, the Flexible Packaging method makes a two-tier distinction



in order to distribute the responsibilities among three participants.

Given our goal of having the packaging specialist encapsulate the low-level packaging details, how should this encapsulation be achieved? To motivate Flexible Packaging's answer, three characteristics of the packaging specialist's job are noteworthy. First, the packaging specialist is responsible not only for the source code needed to achieve a given packaging, but also for the necessary non-code artifacts (like IDL files) and the construction and installation steps. Second, the products for which the packaging specialist is responsible are not fixed, but vary according to the high-level packaging decisions that the system architect makes. For instance, with ActiveX, the system architect chooses the COM interfaces that the control exports, as well as the methods and properties that constitute those interfaces. The packaging specialist is then responsible for producing the ATL source code and other products that reflect the system architect's COM interface choices. The packaging specialist encapsulates the syntax and tools associated with these products, but must allow the content to vary with the architect's decisions. Third, because the system architect 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 packaging specialist is responsible cannot be produced until then.

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 architect's high-level packaging decisions. Modules with parameters (like SML functors or Ada generics) are capable of capturing the variability in this source code (based on my experience building packaging generators) but are not applicable to the non-code 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, non-code artifacts, and construction and installation steps that are needed to achieve the packaging.

The parameters that capture the system architect's essential packaging decisions are written in the form of a packaging description. An example packaging description, shown previously in Chapter 1, describes an ActiveX control for displaying PNG images:

```
PNGViewerControlInterface: interface ActiveXControlPackaging with
   library_iid: UUID{7CF18AA0-36FF-11d2-9FD5-00104B33709D};
   coclass_iid: UUID{7CF18AA1-36FF-11d2-9FD5-00104B33709D};
   name: "PNGViewerControl";
   help_string: "PNG Image Control for DeLine's Thesis";
   IPNGViewerControl: player COM_Interface with
      iid: UUID{7CF18AA2-36FF-11d2-9FD5-00104B33709D};
      help_string: "PNG Image Interface";
      FileName: player COM_Property with
         id: 1;
         signature: "BSTR";
      end
   end
end
```

This packaging description is really a compound parameter, like a record in a standard programming language. From this description, the ActiveX generator produces the ATL source code, IDL file, registry file, and resource file that were discussed in Section 3.1.1. The generator also produces a Makefile (set of construction steps) that processes the source code and non-code artifacts. As described in Section 4.2, the Flexible Packaging method provides a set of Java classes for creating these packaging generators.

In summary, achieving a particular component packaging involves both making high-level packaging decisions and deriving detailed artifacts from these decisions. The Flexible Packaging method places the responsibility for the high-level packaging decisions in the hands of the system architect. To keep the architect from being overburdened, the method introduces the role of the packaging specialist, who provides a software generator to map the architect's high-level decisions to the detailed artifacts. Later, Section 4.2 describes how the packaging specialist creates this generator.

3.2 MIXING WARES AND PACKAGERS TO PRODUCE COMPONENTS

3.2.1 *A component is a ware plus a packager*

To review, when using the Flexible Packaging method to create a component, the system architect first selects a ware that implements the desired functionality. Next, she writes a packaging description that captures the desired packaging and runs a tool called Packgen, which automatically selects the appropriate packaging generator to run on the packaging description. The packaging generator creates the source code, non-code artifacts, and construction and installation instructions needed to achieve the packaging.

The source code that the packaging generator produces is called the packager. The ware and the packager are complements: the ware implements the component's functionality; the packager, its interactions with other components. The ware's and packager's computations are combined to form the component's entire run-time behavior. How should this combination be achieved?

The mechanism chosen for combining the ware and the packager needs to support the ability to mix-and-match these two modules. The goal of Flexible Packaging is to allow a ware to be packaged in multiple ways, which means that a ware needs to be combinable with different packagers generated from different packaging descriptions. Conversely, a packager should be combinable with different wares. There are potentially many pieces of functionality that one would like to make available, for example, as ActiveX controls, which means combining an ActiveX packager with many different wares.

The goal, it should be noted, is not to support the arbitrary mix-and-match of wares and packagers: every piece of functionality cannot be made available through every packaging. Some combinations of functionality and packaging simply do not make sense. For instance, the ware introduced in Chapter 1, which parses and paints PNG images, cannot be packaged as a Unix filter. The ware requires the ability to paint to a window, which a (pure) Unix filter cannot provide, since Unix filters are for text stream transduction. Further, evolving a software component to have a new style of interaction can take considerable effort.[15] The goal of Flexible Packaging, then, is to support a reasonable amount of mix-and-match between wares and packagers and to detect ware/packager mismatches like that of the PNG Unix filter. The question of what constitutes a reasonable amount of mix-and-match is considered later in Section 3.2.5.

In addition to supporting mix-and-match, whatever mechanism is chosen for combining wares and packagers should not place an undue burden on the software architect. In the traditional approach to system integration, the system architect creates glue code to overcome packaging mismatches between components. If combining a ware and a packager requires as much effort as creating glue code to combine components, the Flexible Packaging method would provide no advantage over the traditional approach. The method would simply have moved the problem of mismatch resolution from between components to within components, without making the problem any easier.

Hence, we need a means of combining a ware and packager that both allows a reasonable degree of mix-and-match and does not require as much integration effort as the original component integration problem. First, I consider having the packager export a traditional interface to the ware and then having the ware export an interface to the packager, neither of which proves to be satisfying. Then, I introduce interfaces based on coroutines, which provides the desired mix-and-match with low integration effort.

3.2.2 *Take 1: The packager exports an API to the ware*

In general, combining a ware and a packager amounts to combining two modules written in a standard programming language. Given this, the most ready way to achieve the combination is to use the mechanisms inherent in the language, like procedure call. Indeed, procedural interfaces (often called application programming interfaces, or APIs) are the basis of much of today's reuse. With a traditional interface, a service provider and a service client agree on a collection of procedures by which the service is invoked. The service provider implements the collection of procedures, and the service client calls the procedures to invoke the service. (Although this section discusses interfaces based on procedures, a similar argument could be made for interfaces based on higher-order functions or methods. The terms interface or API will be used throughout to remain neutral about the mechanism.) Using an agreed-upon interface to integrate software modules is called programming-by-contract, particularly when the benefits and obligations inherent in the procedures are made clear through pre- and postconditions.[36] The idea of building a computation out of another computation with an explicit interface is the basis of many of computer science's most successful ideas: layered systems;[17] information hiding;[39] data abstraction and abstract datatypes;[24] and formalisms for reasoning about composition, like Hoare logic[23] and algebraic interface specifications.[22]

As useful as traditional APIs are for building many classes of software systems, they are unfortunately ill suited to our goal of mixing different wares with different packagers. An API, as a form of interaction between modules, binds too many decisions and thus hinders mix-and-match. There are two ways to attempt to achieve our desired mix-and-match using a traditional API: we could consider the packager as the service provider that exports an API to the ware or vice versa. We will consider each of these possibilities in turn. Because there are far more potential wares than packagers (there are more potential bundles of functionality than there are ways for components to interact), let's start by considering the packager as the service provider.

With the packager as the service provider, each packager would come with a procedural interface against which wares are written. (This is essentially the traditional idea of abstract I/O libraries.[40]) This approach makes combining a packager with different wares straightforward: each ware would be written against the packager's procedural interface, and the system architect would combine the ware and the packager by linking. Unfortunately, the converse – combining a ware with different packagers – would be problematic because the various packagers' interfaces would differ.

To appreciate the differences that could arise between different packagers' interfaces, consider two example components that compute the mean of a set of integer data. In Figure 3.4(a), the first component pulls the integer data from a file and pushes the mean to standard output. In Figure 3.4(b), the "outside world" sends the second component messages, pushing both the integer data and a request for the mean to the component. The figure illustrates a reasonable approach to implementing each of these components as a packager and a ware that coordinate through a procedural interface. Each packager's interface consists of both procedures that the packager exports to the ware and procedures that the ware is obliged to export to the packager. (The latter could also be handled through function pointers – so-called call-back functions – but that does not affect the argument here.)

I designed each interface with only its own component in mind; hence the two interfaces are very different from each other. In the two interfaces, the data push/pull distinction is reflected in the direction of the procedure calls: when the data is pulled, the packager defines procedures for the ware to call; when the data is pushed, the ware defines procedures for the packager to call. The reason is that a procedure call represents a particular pattern of data and control flow, and the packager's interface encodes the component's interactions into this pattern. This encoding is particularly awkward (though typical) for the message-pass-

FIGURE 3.4 The implementations of two components that calculate the mean of a series of integers. (a) The component gets its data from a file and reports its result to standard output. (b) The component gets its data from messages and reports its result through a message.

(a) *Module P- le*

```
int MoreData()
{
   return ! eof(F);
}

int GetData()
{
   int i;
   fscanf(F, “%d”, &i);
   return i;
}

void main()
{
   F = fopen(DataFile, “r”);
   int result = RunWare();
   printf(“mean %d\n”, mean);
}
```

Module W- le

```
int RunWare()
{
   int n=0, sum=0;
   while (MoreData()) {
      sum += GetData();
      n++;
   }
   return sum/n;
}
```

(b) *Module P-message*

```
void DataCB(message *m)
{
   PutData(m->data);
}

void MeanCB(message* m)
{
   m->reply = GetResult();
}

void main()
{
   InitMessageSystem();
   RegisterCallback(M1, DataCB);
   RegisterCallback(M2, MeanCB);
   WaitForMessages();
}
```

Module W-message

```
int n=0; sum=0;

void PutData(int i)
{
   sum += i;
   n++;
}

int GetResult()
{
   return sum/n;
}
```

ing component, since the ware's functionality is fragmented across different procedure definitions.

Two approaches could be used to cope with the differences between different packagers' interfaces: standardizing the packagers' interfaces to eliminate the differences or writing glue code to compensate for the differences. In the first approach, the packaging specialists would agree to make their packagers export a standardized interface. Having all packagers export the same interface would allow the packagers to be substitutable for one another, and hence the system architect could readily link a ware with any packager that implements the standard. Unfortunately, this kind of cross-industry cooperation is difficult to achieve, especially over the course of time. This approach also requires the services that the packagers provide to be sufficiently similar to be united under a single interface, which not plausible given the effect that the data push/pull distinction has on the direction of the procedure calls in the interface.

The second approach – writing glue code – is more promising. A ware would initially be written against a particular packager's procedural interface, and a system architect would combine that ware and that packager by linking. To combine that ware with a different packager, the system architect would create the necessary glue code to overcome the differences between the original packager's procedural interface and the new packager's procedural interface. The architect would then interpose the glue code between the reused ware and the new packager. For instance, Figure 3.5(a) shows the glue code that integrates *P-message* and *W- le*; Figure 3.5(b), the glue code that integrates *P- le* and *W-message*. As Black points out, the glue code between two active computations, like *P- le* and *W-message*, must act as a buffer; the glue code between two passive computations, like *P-message* and *W- le*, must act as a pump.[4] In both cases, the glue code is not negligible and performs a significant computation.

In short, it is dubious to treat the packager as a service provider that exports a procedural interface to the ware. Making the packager be the service provider either requires an unlikely agreement among packager interfaces or requires the system architect to create glue code to overcome the differences between one packager's interface and another's.

3.2.3 *Take 2: The ware exports an* API *to the packager*

The converse option – considering the ware to be the service provider and the packager to be the service client – is even more difficult. First, the service that a packager provides is always about component interaction; the service that a ware provides could be about any subject matter whatever (mathematical calculation, calendar management, stock market prediction, vehicle navigation, word processing). Whereas there might be some chance of establishing standard APIs among packagers, there is no chance of such standards between wares. For the same reason, creating glue code to overcome the difference's between one ware's API and another's is more difficult than the analogous job for packagers.

FIGURE 3.5 The glue code needed to overcome the differences between packagers' interfaces. (a) The glue code needed to integrate *P-message* and *W- le*. (b) The glue code needed to integrate *P- le* and *W-message*.

(a) *Module Glue-bu er*

```
linked_list Buffer = create_list();

void PutData(int i)
{
   list_append(Buffer, i);
}

int GetResult()
{
   return RunWare();
}

int MoreData()
{
   return ! list_empty(Buffer);
}

int GetData()
{
   int result = list_head(Buffer);
   Buffer = list_tail(Buffer);
   return result;
}
```

(b) *Module Glue-pump*

```
int RunWare()
{
   while (MoreData())
      PutData(GetData());
   return GetResult();
}
```

Second, one might object and claim that, because packagers are generated, the generator could consider the ware's interface when generating the packager, thereby obviating the need for glue code. However, given the nature of the potential differences between different wares' interfaces, creating a generator with enough "smarts" to overcome such differences would be difficult and expensive. To support this smart generation, the description of the ware's procedural interface would have to be much more explicit and detailed than current interface descriptions. Hence, having the ware provide an API to the packager is even more dubious than the reverse.

In summary, a traditional interface is ill-suited for our intended mix-and-match of wares and packagers, whether the packager exports an interface to the ware or vice versa. Instead, Flexible Packaging introduces a new type of interface between these two modules, an interface which is based on a construct called a channel and which better supports our intended mix-and-match.

3.2.4 *Take 3: The ware and packager export channels to one another*

Given the problems associated with having a traditional API between the packager and ware, the Flexible Packaging method promotes a modularity mechanism based on the old idea of coroutines.[11] Rather than interacting through procedure calls, function calls, or method calls, the ware and the packager interact through strongly typed unidirectional data streams, called channels, and the execution of the ware and packager is interleaved through coroutines. For a long time, developers have known that the use of coroutines between modules allows those modules to have independent control structures. (Indeed, the original purpose for which Conway invented coroutines was to allow a compiler's lexer and parser to have independent control structures.) The Flexible Packaging method takes advantage of this property of coroutines to reduce the restrictions that a ware or packager makes on its partner-module, thereby promoting mix-and-match between wares and packagers.

The Ciao and Ciao++ language constructs To use this channel construct, the source code for wares and packagers is written in either an extension of C, called Ciao, or an extension of C++, called Ciao++. Both Ciao and Ciao++ add the same language constructs to their respective base languages: *channel*; *in*; *out*; and *alt*. These constructs are syntactic sugar for library calls. The translation from this syntactic sugar to the library calls provides an opportunity to type-check the use of the constructs, a step that would be unnecessary in a language with polymorphic types. Each of the four channel constructs is described below.

Although channels in C++ could have been implemented with templates, I used the syntactic sugar approach for consistency with C.

channel [**in**|**out**] [**stream**] <*type*> <*chname*>**;**

A *channel* construct declares a channel, giving it a direction (*in* or *out*), an arity (*scalar* or *stream*), a type, and a name. A scalar channel is used to communicate a single value; a stream channel is conceptually a queue: multiple values may be put on a stream channel before they are consumed. A stream channel supports three functions that a scalar channel does not: *open*, which allows a computation to place values on a stream channel after it has been previously closed; *close*, which signals that no more values are to be placed on a stream channel; and *more*, which checks a stream channel has not yet been closed.

Because scalar channels are buffered, as described in Section 3.2.5, both scalar and stream channels share the same implementation. The scalar/stream distinction allows a programmer to document his intended use of the channel.

in(<*chname*>, <*variable*>)**;**

An *in* statement gets a value off a channel. The channel must be declared with direction *in*, and the type of the variable must be the same as the declared type of the channel.

out(<*chname*>, <*expression*>)**;**

An *out* statement puts a value on a channel. The channel must be declared with direction *out*, and the type of the expression must be the same as the declared type of the channel.

alt {
    <*in-statement*>: <*statement*>
    <*in-statement*>: <*statement*> …
    <*in-statement*>: <*statement*>
}

The *alt* construct, like its namesake in occam,[26] allows an input to occur from any one of a set of *in* statements for which input is ready. After the selected *in* statement occurs, the statements following the colon are executed.

When more than one of the *in* statements in an *alt* statement has data ready, the current implementation chooses the first one listed in the *alt*. This situation only arises because *out* statements are buffered. A better implementation might be to select the *in* statement whose data is "oldest," i.e. buffered for the longest time. The Ciao compiler's deadlock check is stronger than necessary: it uses non-determinism to detect the possibility of deadlock, whatever the scheduling policy for *alt* statements. Because of the strength of this check, the compiler may report false positives for deadlock.

The use of channels to coordinate modules has both a data-flow and a control-flow aspect. In terms of data flow, when one module does an *out* statement on a channel and another module does an *in* statement on the same channel, the value flows from the *out* statement to the *in* statement. Explicitly, there is an assignment of the expression in the *out* statement to the variable in the *in* statement. (For channels with record types, the assignment is field-wise.) Since this data flow has the same meaning as assignment, *in* and *out* statements on channels of array or pointer types cause aliasing, just as assignment does.

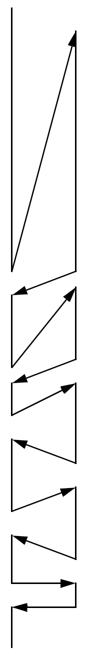
The control-flow aspect of channels is like coroutines: after a module executes an *in* or *out* statement, the thread of control leaves that module and resumes execution wherever it left off in the other module. The control-flow aspect of channels is illustrated in Figure 3.6. The component in the figure consists of two Ciao modules: the packager in module *P*; and the ware in module *W*. The component is packaged as a batch program; hence, obeying the rules of this packaging, module *P* exports a function called *main*. How the thread of control enters the component is packaging-specific. For a batch program, the thread of control begins in

FIGURE 3.6 A small example of the use of channels to coordinate modules. The arrows indicate how the thread of control bounces back and forth between the modules in the style of coroutines.

```
void main()
{
   channel out stream int N;
   channel in int Min;
   channel in int Max;
   int i;

   scanf("%d", &i);
   out(N, i);
   scanf("%d", &i);
   out(N, i);
   scanf("%d", &i);
   out(N, i);

   in(Min, i);
   printf("Minimum is %d", i);
   in(Max, i);
   printf("Maximum is %d", i);
}
```



```
coroutine void Compute()
{
   channel in stream int N;
   channel out int Min;
   channel out int Max;
   int n, min, max;

   in(N, n);
   min = n;
   max = n;
   in(N, n);
   if (n < min) min = n;
   if (n > max) max = n;
   in(N, n);
   if (n < min) min = n;
   if (n > max) max = n;
   out(Min, min);
   out(Max, max);
}
```

the function *main*. This function executes until it reaches the *out* statement on channel *N*. The thread then resumes execution where it left off in the module *W*. In this case, because the thread has not yet executed any code in module *W*, it starts at the beginning of the function declared to be a coroutine, *Compute*. The thread then executes until it reaches the *in* statement on channel *N*. After the data value is transferred between the *out* statement on *N* and the *in* statement on *N*, the thread resumes execution where it left off in module *P* (just after the *out* statement). In this way, the thread of control bounces back and forth between the modules on every occurrence of an *in* or *out* statement, and the modules' computations are effectively interleaved.

The addition of these constructs to C and C++ do not significantly impact the semantics of the language. First, these constructs are type safe. As mentioned earlier, the type of an expression in an *out* statement must exactly match the declared channel type. Similarly, the type of a variable in an *in* statement must exactly match the declared channel type. Hence *in* and *out* statements cannot cause type-unsafe data flows. The Ciao and Ciao++ compilers check for type violations. In terms of run-time behavior, the Ciao constructs do not affect the languages' other control flow constructs, including interrupts and *setjmp/longjmp*. The exception is program termination. When the Ciao run-time implementation detects deadlock, it uses a call to *exit* to halt the program with an error message. (In the absence of deadlock, the program terminates in the usual way.) Because the coroutines are implemented using Microsoft's Fiber (light-weight thread) library, Ciao and Ciao++ modules themselves should not make calls to this library.

Channels support independent control structures The advantage of interfaces based on coroutines is that the modules are free to have whatever internal control structure is convenient. When modules coordinate through a procedural interface, the interface both serves as a contract between the modules and dictates the functional decomposition of the module implementing the service. This dual role of a traditional interface is evident in the message-passing component in Figure 3.4(b). The interface between the packager and the ware allows the packager to push data to the ware, but forces the ware's functionality to be awkwardly distributed among different procedures. Also, as previously discussed, the data push/pull distinction influences the direction of the procedure calls in a packager's interface to a ware. Hence, the two packager interfaces in Figure 3.4 are quite different from one another.

In contrast, Figure 3.7 shows the same interface implemented using channels. With this channel-based interface, the ware *W* may be readily integrated with either the file packager *P- le* or the message packager *P-message*, with no need for glue code. Whereas the data push/pull distinction affects the direction of the procedure calls in procedure-based interfaces, this distinction is not manifest in channel-based interfaces. As a result, the interfaces to *P- le* and *P-message* are the same:

```
Pkg = out(N) → Pkg [?] in(Mean) → DONE
channel out stream int N
channel in int Mean
```

despite the fact that *P- le* receives its data via pull and *P-message* receives its data via push. Also, the ware *W*'s source code is written as a single coherent procedure, whether integrated with *P- le* or *P-message*. Because the module interaction is done through channels rather than procedures, how the modules interact no longer affects how each module is decomposed into procedures.

3.2.5 *Additional channel features*

As the example components in Figure 3.7 illustrate, basing the interface between the ware and packager on coroutines rather than procedure calls makes substantial progress toward our desired mix-and-match. However, with the channel mechanism described so far, ordering mismatch, name mismatch, and data representation mismatch can still create incompatibilities between the ware and packager. Channel buffering and channel maps, explained in this section, address these mismatches. With these final channel features in place, I then describe how the channel mechanism allows a reasonable amount of mix-and-match between

FIGURE 3.7 The two components from Figure 3.4, re-implemented using channels.

Module W

```
coroutine void RunWare()
{
    channel in stream int N;
    channel out int Mean;
    int i, n=0, sum=0;
    while (more(N)) {
        in(N, i);
        sum += i;
        n++;
    }
    out(Mean, sum/n);
}
```

Module P- le

```
void main()
{
    channel out stream int N;
    channel in int Mean;
    int i, m;
    F = fopen(DataFile, "r");
    open(N);
    while (! eof(F)) {
        fscanf(F, "%d", &i);
        out(N, i);
    }
    close(N);
    in(Mean, mean);
    printf("mean %d\n", mean);
}
```

Module P-message

```
channel out stream int N;
channel in int Mean;

void DataCB(message *m)
{
    out(N, m->data);
}

void MeanCB(message* m)
{
    int mean;
    close(N);
    in(Mean, mean);
    m->reply = mean;
}

void main()
{
    open(N);
    InitMessageSystem();
    RegisterCallback(M1, DataCB
    RegisterCallback(M2, MeanC
    WaitForMessages();
}
```

wares and packagers and how the system architect determines ware/ packager compatibility.

Alleviating ordering mismatch The use of channels, as discussed so far, is restrictive in that the modules must agree on the exact order in which the channels are used. For example, when one module executes an *in* (*out*) statement on a given channel, the other module must then execute an *out* (*in*) statement on the same channel. To loosen this restriction, an *out* statement's implementation buffers the data provided in the *out* statement until the corresponding *in* statement consumes the data. Because of the buffering, execution may proceed past an *out* statement without the execution of corresponding *in* statement; however, an *in* statement must still block until the corresponding *out* statement is executed, that is, until the channel's buffer is not empty. In the presence of buffering, deadlock is now possible: each module may be stuck at an *in* statement, requiring data from the other module in order to proceed. Hence, for two modules to coordinate correctly at run time, rather than agreeing on the exact order in which channels are used, the modules instead must avoid deadlock. How deadlock is prevented at compile time and detected at run time is discussed in Chapter 4. An implication of this weakened condition for module coordination is that the value that an *out* statement puts on a channel need not ever be consumed by an *in* statement. Indeed, an *out* channel in one module need not have a corresponding *in* channel in the other module.

An *out* statement without a corresponding *in* statement introduces the possibility of a memory leak, which the current Ciao compiler does nothing to address. When either an *out* channel has no corresponding *in* channel or when the source code contains no *in* statements on the corresponding *in* channel, then there is no possibility that the *out* statement's value will be consumed. In this case, as an optimization, the compiler could eliminate the *out* statement. In the general case, it is undecidable whether a given *out* statement leaks memory.

Alleviating name and data representation mismatch Of the possible differences in channel-based interfaces, differences in data representation and connection establishment are likely to be quite common. That is, without prior agreement or conventions, two independent programmers are quite likely to choose different names and types for the same channel. Given this, the Ciao compiler, which the system architect uses to combine a ware and a packager, simplifies the ability to overcome these differences. As mentioned in Chapter 1, when the system architect combines a ware and a packager, he hands the compiler the ware's source code, the packager's source code, and a table called a channel map. An example channel map, shown earlier in Chapter 1, is the following:

```
Initialize    Begin

NewFile       SetFileName      channel in BSTR windows_string;
                               channel out char* c_string;
                               BSTR bstr;
                               char* cstr = (char*)malloc(100);
                               in(windows_string, bstr);
                               sprintf(cstr, \"%S\", bstr);
                               out(c_string, cstr);

Finalize      Quit
```

Each entry in the channel map contains the name of a channel declared in the ware and the name of the corresponding channel declared in the packager. When the two channels have different types, the map entry may additionally contain source code to convert from the *out* channel's type to the *in* channel's type. If the two channels share a common name but differ in type, the map entry contains the same name twice plus the conversion code.

When a channel's name does not appear in the map, either (1) the channel has a corresponding channel in the other module with the same name and same type or (2) the channel is an *out* channel and there is no corresponding *in* channel in the other module. In the absence of name and type mismatch, the correspondence between channels is based on name matching, like the matching of a procedure caller to a procedure definer with a conventional linker. Because the compiler falls back on name matching, the cost of channel maps is incremental. The more disagreement there is between the ware author and the packager author, the more entries in the channel map. When the ware and packager authors collaborate to agree on channel names and types, the system architect can forgo the channel map.

The Flexible Packaging method currently provides no support to the system architect for overcoming differences in three aspects of interaction: state persistence, state scope, and failure. Chapter 6 discusses the possibility of adding support for these aspects.

Supporting "reasonable" mix-and-match As mentioned earlier, some combinations of functionality and packaging do not make sense; hence, some combinations of wares and packagers do not make sense. Because channels are used to combine a ware and a packager, they are also used to determine the compatibility between them. Each Ciao module (each ware and each packager) comes with a channel signature that lists the module's channels and captures the order in which *in* and *out* statements are executed on those channels. As an example, here is the channel signature for the ware discussed in Chapter 1:

```
Ware =  in(Initialize) → Loop.
Loop =  in(NewFile) → Loop
     [] in(Paint) → Loop
     [] in(Finalize) → DONE.

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

The first half provides a CSP expression to capture the order of channel use; the second half lists the channels.

A ware's channel signature abstractly captures its interaction requirements; a packager's channel signature captures the "shape" that the functionality must have to be compatible with the packaging. As a simple example, consider two wares and two flavors of Unix filter. The first ware, Rev separately reverses each line in a sequence of lines and has the following channel signature:

Rev = in(Line) $\longrightarrow$ out(Backward) $\longrightarrow$ Rev [] DONE.

channel in char* Line;
channel out char* Backward;

The second ware, Sort sorts all of its lines of input and has the following channel signature:

Sort = in(Line) $\longrightarrow$ Sort [] Report.
Report = out(Sorted) $\longrightarrow$ Report [?] DONE.

channel in char* Line;
channel out char* Sorted;

Given these signatures, it is clear that the first ware provides a line of output for every line of input it receives; whereas, the second ware consumes all its input before producing any output.

Now consider two variations on Unix filters. The first is an incremental filter, like the common utilities *cat*, *head*, and *grep*, and its packager has the following channel signature:

Inc = out(I) $\longrightarrow$ in(O) $\longrightarrow$ Inc [?] DONE.

channel out char* I;
channel in char* O;

The second is a non-incremental filter, and its packager would have the following channel signature:

NonInc = out(I) $\longrightarrow$ NonInc [?] Gather.
Gather = in(O) $\longrightarrow$ Gather [] DONE.

channel out char* I;
channel in char* O;

As these signatures reveal, the first packager is compatible with an incremental ware, like Rev, but not a batch ware, like Sort. Indeed, if this packager were combined with Sort, the result would be deadlock: the ware would be stuck on *in*(*Line*); the packager would be stuck on *in*(*O*).

On the other hand, the second packager is readily compatible with a non-incremental ware, like Sort, and also, due to channel buffering, with an incremental ware, like Rev.

Determining ware/packager compatibility Given the task of determining whether a given ware and packager are compatible, the system architect can proceed in three stages. First, she can use her intuition, based on a glance at the ware's signature and the type of the packaging. For instance, one of the biggest differences between packagings is in whether the packaging allows the outside world or the component itself to choose what computation the component performs. An example of the former kind is an ActiveX control, where other components determine what computation the control performs by calling its COM methods. An example of the latter kind is a batch program, which is free to perform whatever computation it chooses. Given this, an ActiveX packager is most likely to be combined with a ware whose signature features external choice at its top level:

$$\text{Ware} = \mathit{service}_1 \; [\,] \; \mathit{service}_2 \; [\,] \; \ldots \; [\,] \; \mathit{service}_k$$

Since other components determine which COM method is called in the ActiveX packager, the packager in turn determines which service the ware performs. On the other hand, a packager for a batch program should not be combined with such a ware, since it will be unable to select which of the services the ware performs. Instead, a ware to be combined with a batch program packager should feature sequencing and internal choice. In this way, the system architect can use her intuition about whether a given ware is well suited to be packaged with a given packaging.

Given that the system architect's intuition suggests compatibility, the next stage is for her to establish a correspondence between the ware's and packager's signatures. At this stage, she may read the channel signatures and realize that there is no correspondence, which means that the ware and packager are incompatible. In particular, the ware and the packager are incompatible if an *in* channel in one has no corresponding *out* channel in the other. For example, were a system architect to try to package the PNG ware as a Unix filter (either incremental or not), she would discover that the ware's *in* channel Paint has no corresponding *out* channel in the filter packager. The ware requires a painting interaction that the filter packager cannot provide.

Once the system architect has established a correspondence between the ware's and packager's signatures, 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. As described in detail in Section 4.3, the Ciao compiler creates a CSP model of the combination of the ware and packager and uses the CSP model checker FDR to checker whether this model successfully terminates or deadlocks. Absence of deadlock in this CSP model implies absence of deadlock in the combined ware and packager, which is the basis of their compatibility.

3.3 WORK RELATED TO FLEXIBLE PACKAGING

Although much of the previous work related to this research appears throughout the dissertation, several projects deserve particular mention.

3.3.1 *Separations of concern*

Gelernter and Carriero were early advocates of the separation of a software component's functionality from its interaction with other components[20]: "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).[9] The Linda model is that 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 an interaction mechanism, shared tuplespace, that sufficiently general to simulate other interaction mechanisms, like RPC and message passing. Rather than advocating a single interaction mechanism, Flexible Packaging allows the system architect 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.[7] 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. How such an inference mechanism could be applied to Flexible Packaging is discussed in Section 6.6.2.

Flexible Packaging is also similar to the work on Aspect-Oriented Programming (AOP) at Xerox PARC.[31] The goal of both research projects is to separate a component's functionality from extra-functional concerns. For Flexible Packaging, the extra-functional concern is component interaction. AOP has explored several different extra-functional concerns, including thread synchronization and data transfer in distributed programs.[32] In their approach, a program written in a standard programming language, like Java, is supplemented with a declarative description of an extra-functional 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 with relative to the functionality. The price to be paid for this reusability is that the system architect must create a channel map to show the relationship between the functional and extra-functional concern, 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, the AOP compiler weaves the source code that implements the functionality together with the source code generated from the aspect at compile time.

3.3.2 *Concurrency and modularity*

The benefits of concurrency on program structure have been studied for a long time. Conway invented coroutines in the early 1960s, for example, to provide a good structure for his COBOL compiler.[11] 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.[29] 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; and the implementation phase, the developer maps the resulting network of concurrent processes onto the available computing and data storage resources.[8] 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."[42] His language Concurrent ML (CML) has been used to create a user interface toolkit based on the X protocol and a distributed programming toolkit.[43]

In these examples, 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 a program's 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 so much because of variable run-time phenomena (like the order of user operations) but because of variable combinations of Ciao modules.

3.3.3 *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 *role*s. 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.

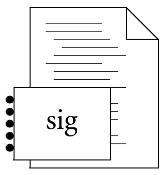
Channel signatures can be thought of as simplified Wright descriptions. A channel signature captures a Ciao module's interactive behavior in the same way that a Wright component's computation section captures how the component's interactive behavior. Indeed, a channel signature can be thought of as the 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 describe a configuration containing the Ciao modules and a channel connector and to run the Wright tools on the configuration, which would then 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.

3.4 FLEXIBLE PACKAGING AT A GLANCE

In summary, this chapter discusses the two key aspects of Flexible Packaging: the introduction of the role of the packaging specialist to prevent burdening the system architect with packaging details; and the use of channels to support the mix-and-match of packagers and wares. With these ideas established, I can now provide an overview of the method as a whole. The role of each participant plays is briefly described below. The tools that each participants uses are mentioned below and discussed at length in Chapter 4.

3.4.1 *The ware provider's job*

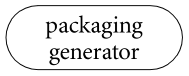
The ware provider creates source code in Ciao or Ciao++ to implement the functionality that he wishes to provide. As part of the ware's development, he chooses those interactions that are flexible and those that are fixed. The flexible interactions are implemented with channels; the fixed interactions are implemented in the traditional way, for example, by calling I/O libraries. For instance, with the PNG ware, the source of the painting context and the file name has been made flexible and is implemented with channels; the interactions that paint pixels to the window are not flexible and are implemented with calls to a standard Windows library. 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 runs a tool that automatically generates a channel signature from the source code. As a last step, he makes the source code and channel signature, pictured at right, available for use.



ware

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 in packaging description that the system architect must provide. This packaging description represent the essential decisions about the packaging. He then uses a framework to create a generator which maps the system architect's packaging description to the source code, non-code artifacts, and construction and installation steps needed to implement the described packaging. His final step is to make the packaging generator, pictured at right, available for use.

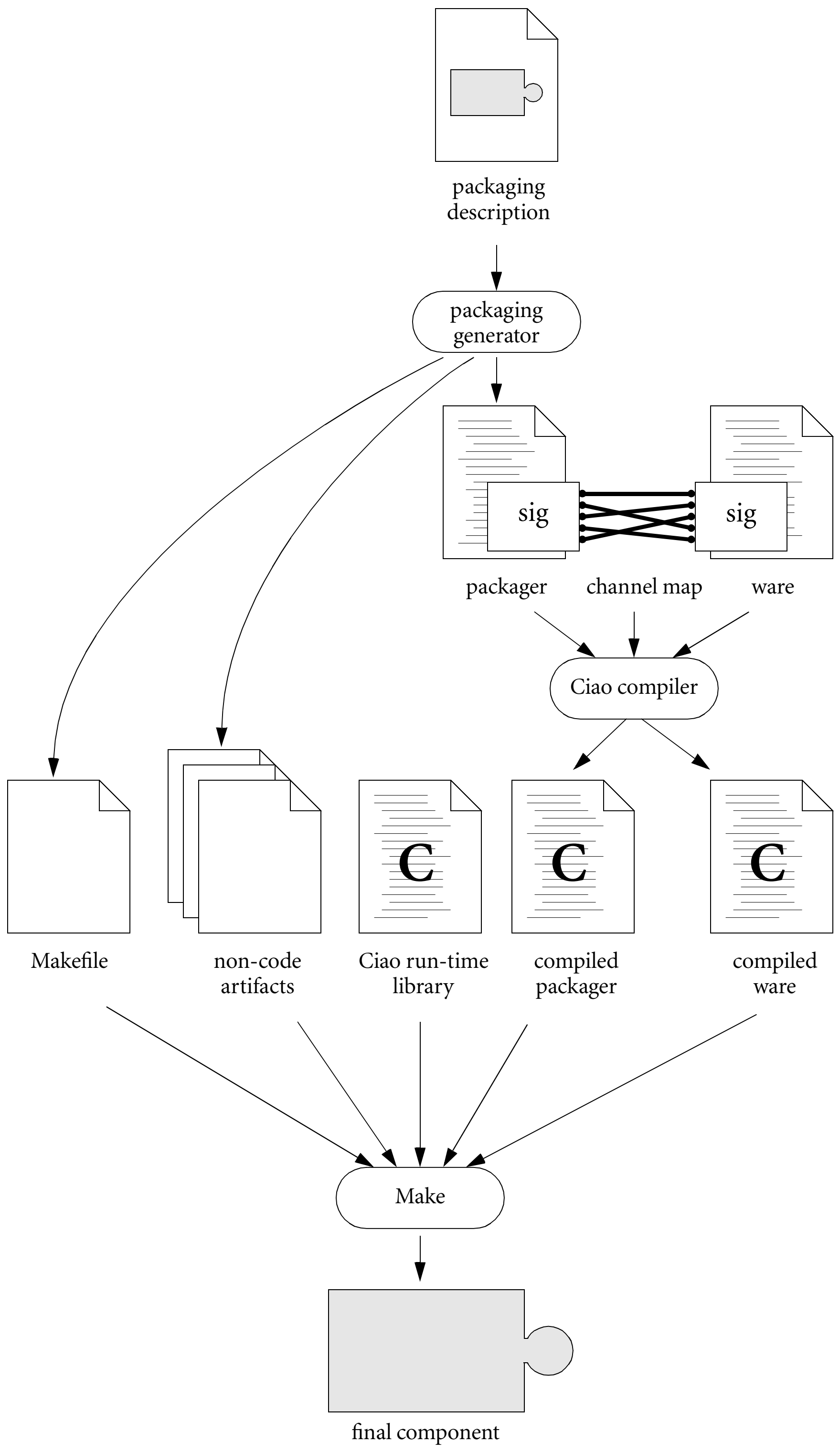


3.4.3 *The system architect's job*

After selecting a ware to use, the system architect turns the ware into a full-fledged component in two steps, pictured in Figure 3.8. 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. She then invokes the Packgen tool on the packaging description, which in turn runs the appropriate packaging generator. The result of running Packgen is a packager with its channel signature, non-code 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.

FIGURE 3.8 The two steps that the system architect performs to produce a component tailored to an integration context. As the first step, she writes a packaging description and runs the tool Packgen on the description. Packgen in turn invokes the appropriate packaging generator, which produces the packager, non-code artifacts, and construction steps (Makefile) needed to achieve the packaging. As the second step, she runs UniCon on a description of her component, which first runs the Ciao compiler on the ware and packager and then runs Make to produce the final component.



4 *Tools for Flexible Packaging*

The previous chapter discussed the main ideas behind Flexible Packaging and provided a general overview of the method. One of the goals in the design of Flexible Packaging was to make each participant's job as painless as possible, in part through a thoughtful distribution of responsibilities across participants, in part through the provision of useful tools. This chapter focuses on the specific tools that each of the participants uses to fulfil his role. Section 4.1 describes SigGen, a tool that the ware provider uses automatically to extract a channel signature from his Ciao source code. Section 4.2 discusses the tools that ease the packaging specialist's creation of a packaging generator: UniCon describes the information that the system architect is required to include in the packaging description; Echo facilitates reading the packaging description; PackagerMaker is the Java class from which the packaging generator, a Java class, must be derived; GlueCode allows a generator to produce new files from templates; and Macro allows a generator to produce a new Ciao or Ciao++ source file by manipulating a syntax tree. Finally, Section 4.3 describes the use of FDR to check channel signature compatibility and the Ciao run-time library.

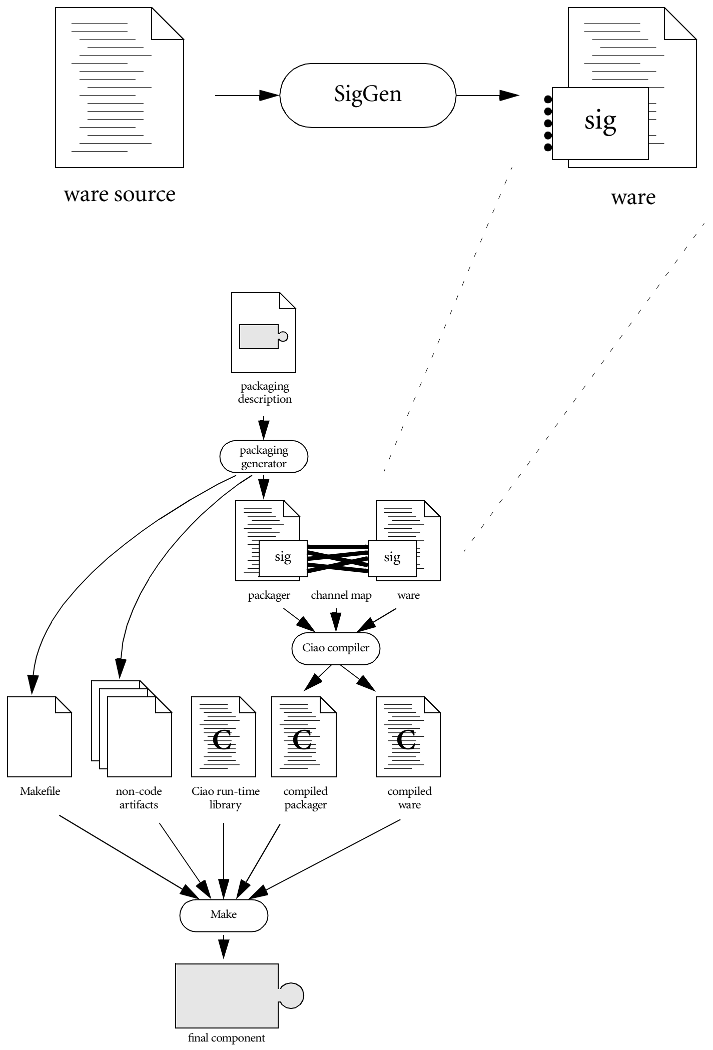
4.1 TOOLS FOR THE WARE PROVIDER

As discussed in the last chapter, the ware provider implements his intended functionality in either Ciao or Ciao++. If writes his code in Ciao, he then runs a tool, called SigGen, which automatically produces a channel signature from the source code, as shown in Figure 4.1. (A version of SigGen for Ciao++ is future work.) This section describes the implementation of SigGen. The Ciao and Ciao++ compilers themselves are described later in Section 4.3.

4.1.1 *Channel signatures*

Before describing how SigGen works, I first define channel signatures. A channel signature is an expression in the following subset of CSP,[25] where *id* is an identifier in the style of C:

FIGURE 4.1 The ware provider runs SigGen on the ware's source code to produce the ware's channel signature.



sig:	*id* = *proc*.	signature definition
proc:	*id*	named process
	in(*id*) $\to$ *proc*	process prefixed with in
	out(*id*) $\to$ *proc*	process prefixed with out
	id ; *proc*	process sequencing
	proc [?] *proc* [?] … [?] *proc*	internal choice
	proc [] *proc* [] … [] *proc*	exernal choice
	DONE	successful termination

A signature definition assigns a name to a process. A process specifies all the orders in which *in* and *out* statements may occur at run time, namely the orders given by the process's trace set.[25]

4.1.2 *Generating channel signatures*

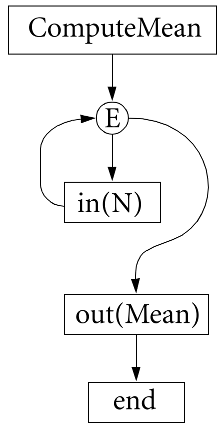
Given a collection of Ciao source files, SigGen iterates over the files, parsing each into an abstract syntax tree (AST). SigGen processes two types of constructs for each AST. First, it processes each type definition (*typedef*) and stores the representation of the type in a table under the type's name. When a channel is declared on a user-defined type, SigGen reports the type of the underlying representation as the channel's type, i.e. SigGen treats user-defined types as transparent types. If the channel types were opaque, the system architect would be unable to detect channel type mismatches when matching the packager's channels with the ware's.

The second AST construct that SigGen processes are function definitions. For each function definition, SigGen creates a control flow graph (CFG) that is conventional with two exceptions. First, since the tool's output is a channel signature, which captures only the order in which *in* and *out* statements take place, the only statements that are considered in the CFG are *in* statements, *out* statements, and those statements that effect control flow; other statements, like assignments that do not involve function calls, are ignored. Second, in addition to conventional (internal) conditional nodes, SigGen's CFG also contains external conditional nodes. External conditional nodes represent changes in control flow that depend only on the state of the channels; internal conditional nodes represent changes in control flow that depend on at least some non-channel state, like local variables. Hence, the nodes that appear in the CFG are of one of the following types: function entry; function exit; in statement; out statement; internal conditional; external conditional; and function call. Figure 4.2 shows a Ciao function definition and the control flow graph that SigGen creates for that function definition. Because the conditional expression in the *while* statement depends only on the state of the channel N, an external node is used to represent the *while*.

From the control flow graph, SigGen computes a process definition according to the recursive translation function T given in Figure 4.3. It then stores this process definition in a global table under the function's

FIGURE 4.2 A Ciao function definition and the control flow graph created from the function definition. Conditional nodes are depicted with circles labeled with either “I” or “E” to distinguish internal from external conditions.

```
coroutine void ComputeMean()
{
    channel in stream int N;
    channel out int Mean;
    int i, n=0, sum=0;
    while (more(N)) {
        in(N, i);
        sum += i;
        n++;
    }
    out(Mean, sum/n);
}
```



name. In addition to using this table to produce the final channel signature, SigGen also uses the table to process function-call nodes. To translate a function-call node, SigGen looks up the function’s name in the table. If the name appears in the table and its process definition consists of more than the process DONE, SigGen uses the process sequencing construct (semicolon) to insert a transition to this function’s process into the process being defined. If the function name does not appear in the table, the function-call node is ignored. This approach assumes that the ware provider has given SigGen all the relevant source code and that functions that appear only as prototypes (e.g. library functions) do not use channels. These assumptions are reasonable given that SigGen is meant to provide a leg up on producing channel signatures; the responsibility for the accuracy of a ware’s channel signature ultimately lies in the hands of the ware provider.

To make the generated signatures more readable, SigGen also limits the repetition of process definitions. In particular, for each node in a CFG that has multiple in-coming edges, SigGen assigns a generated name to that node’s process definition and, where needed, repeats the name rather than the whole definition. For example, given the following function definition

```
void F()
{
    int i, j = random() % 10;
    if (j > 5) in(A, i); else in(B, i);
    out(C, i);
    out(D, j);
}
```

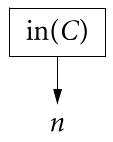
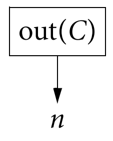
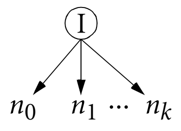
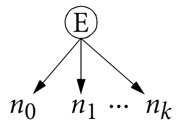
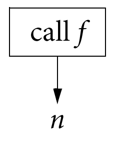
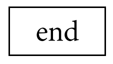
	$\mathrm{in}(C) \rightarrow \mathrm{T}(n)$
	$\mathrm{out}(C) \rightarrow \mathrm{T}(n)$
	$\mathrm{T}(n_0)$ [?] $\mathrm{T}(n_1)$ [?] … [?] $\mathrm{T}(n_k)$
	$\mathrm{T}(n_0)$ [] $\mathrm{T}(n_1)$ [] … [] $\mathrm{T}(n_k)$
	$\begin{cases} f;\, \mathrm{T}(n) & \text{if } f \text{ is in the table} \\ \mathrm{T}(n) & \text{otherwise} \end{cases}$
	DONE

FIGURE 4.3 The translation function T from a control flow graph to a process definition. A successor node of a given node is denoted by *n*.

SigGen will generate the following signature

F = in(A) → F0 [?] in(B) → F0.
F0 = out(C) → out(D) → DONE.

rather than the more verbose signature

F = in(A) → out(C) → out(D) → DONE
    [?]in(B) → out(C) → out(D) → DONE.

For readability, these generated process names are derived from the function name, but the current implementation of SigGen does not prevent the generated names from clashing with user-defined names.

SigGen's output consists of two parts. First, it lists the channel definitions that it encountered. Second, it lists the relevant process definitions, starting with those functions declared as *coroutine* and including the functions transitively called from the *coroutine* functions.

4.1.3 *Accuracy of the generated channel signature*

Automatically generating a channel signature from the ware's source code has two effects that the ware provider may find objectionable. First, because of the approximate nature of a control flow analysis, the generated signature may not accurately reflect the order in which the ware does *in* and *out* statements on its channels. Such an inaccuracy crops up with the PNG ware. The ware's coroutine contains the following loop

```
while (!done) {
    alt {
        in(NewFile): ...
        in(Paint): ...
        in(Done): done = 1;
    }
}
```

from which SigGen generates the following channel signature

```
Loop =
      ( in(NewFile) → Loop [] in(Paint) → Loop [] in(Done) → Loop )
      [?] DONE
```

Because SigGen does not take data flow into account, it is unable to detect the relationship between the *in* on channel *Done* and the termination of the loop. The following channel signature accurately reflects the code's content:

```
Loop = in(NewFile) → Loop [] in(Paint) → Loop [] in(Done) → DONE
```

Currently, the ware provider would have to inspect the generated channel signature, notice the inaccuracy, and correct it by hand. Improving SigGen to take data flow into account is future work.

However, a more accurate code analysis would not correct the ware provider's possible second objection: the generated channel signature directly reflects the content of the code. The ware's channel signature is its interface description and advertises the ware's capabilities to those who would reuse it. As such, the ware provider may not want to make promises about the ware that are based on the ware's current implementation. For example, if the ware's code happens to do an *in* on channel *A* and then an *in* on channel *B*, the signature will reflect this

```
in(A) → in(B) → P
```

If this order is not fundamental to the ware's functionality, but merely a happenstance of the current code, then the ware provider may want to make a looser promise about ordering, for example

$$\text{in}(A) \rightarrow \text{in}(B) \rightarrow P \quad [?] \quad \text{in}(A) \rightarrow \text{in}(B) \rightarrow P$$

Again, inspection of the generated channel signature is the ware provider's recourse to this problem.

If the ware provider changes the channel signature in order to give away fewer implementation details, he is nonetheless responsible for keeping the signature accurate. In particular, it must be the case that the set of possible orders in which the ware can do *in* and *out* statements at run time is a subset of the set of possible orders that the channel signature describes. The channel signature is free to describe orders that cannot actually happen at run time, but the ware must not exhibit an order at run time that the signature does not include. If there are orders of *in* and *out* statements that the ware can perform at run time that are not described in its channel signature, then there are possible sources of deadlock that the analysis in Section 4.3.1 will not catch. In brief, an inaccurate channel signature is a source of undetected deadlock.

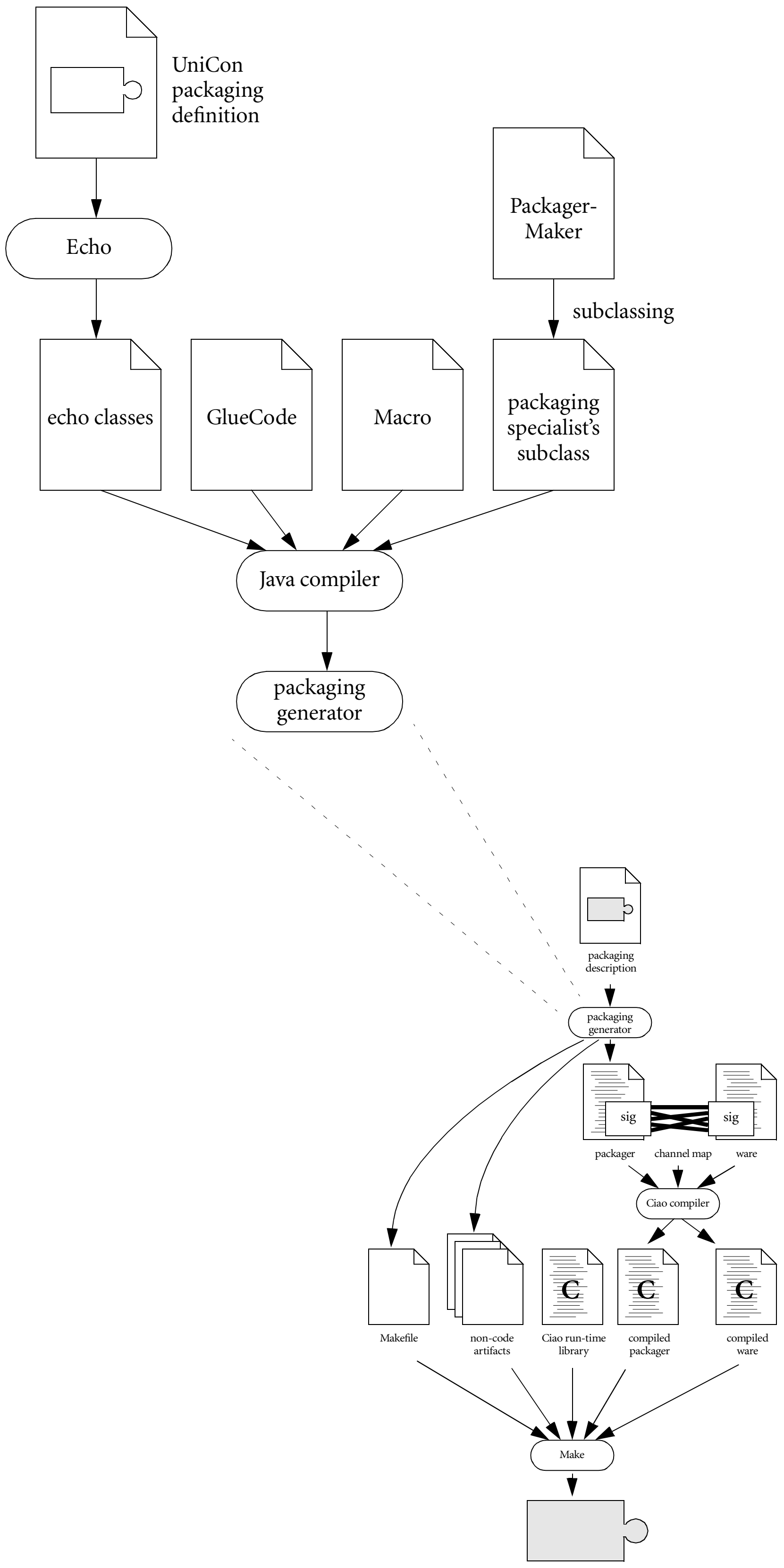
Given the difficulty of automatically producing a satisfactory channel signature from the code, this approach may seem unpromising. However, compare this with today's module systems: the programmer is responsible for producing by hand both the module's implementation and its interface; the module system's tools (e.g. the compiler) checks for consistency between the two. For example, a Modula 3 programmer creates both a module and its interface; an SML programmer creates both a structure and its signature; and a COM programmer creates both a control's source code and its IDL file. A channel signature, however, contains ordering information that these other interface descriptions do not. Producing an accurate channel signature with its ordering information is more difficult than producing an interface description in these other notations. Hence providing tools to lighten the burden of creating a channel signature is appropriate.

4.2 TOOLS FOR THE PACKAGING SPECIALIST

The packaging specialist's job is to capture his expertise about a given packaging in the form of a software generator. This packaging generator accepts a packaging description, from which it generates the Ciao/Ciao++ source code, non-code artifacts, and construction and installation instructions needed to achieve the described packaging.

This section describes the tools that I provide to facilitate the creation of the packaging generator. These tools are illustrated in Figure 4.4. The packaging specialist first writes a definition in the UniCon notation that captures the information to be included the packaging description. He then writes the source code for the packaging generator, which consists of two Java classes, a class of type PackagerMaker and a class of type

FIGURE 4.4 The packaging specialists's tools. The packaging specialist creates a packaging generator, which is a subclass of the Java class PackagerMaker. He may use the classes Macro or GlueCode, if convenient, to generate code and non-code artifacts. He may use Echo to produce Java classes that facilitate reading packaging descriptions written in UniCon.



ConfigurationExpert. When he implements these classes, he may find several Java classes useful: Echo classes are data structures that contain the contents of the packaging description; GlueCode generates new text files from template files; and Macro generates new Ciao source files from template abstract syntax trees. Each of these tools is described in turn.

4.2.1 *UniCon captures packaging abstractions*

A system architect writes her packaging description in the architectural description language UniCon.[47] This notation is well suited for expressing packaging descriptions for two reasons. First, UniCon is an open-ended property-based notation and hence is flexible enough capture whatever parameters are needed to drive the generation. Second, the language is centered around architectural abstractions, like components and connectors, which are the bread and butter of the system architect's job. This means that the system architect can use the same notation to describe both her components' packagings and how the components are assembled with connectors to form the final system. The fact that UniCon is centered on architectural abstractions makes it more appropriate for recording packaging descriptions than more popular open-ended notations, like XML.

Briefly, UniCon describes a software system as a configuration of components and connectors. A component implements a part of the system's functionality; a connector mediates the interaction among components. A component has an interface and an implementation. A component's interface describes its expectations about interaction with other components – what is called its packaging in this dissertation. Its implementation describes the parts out of which the component is built, sometimes a single atomic part (for example, a source file, a library, or an executable), sometimes a configuration of other components and connectors. Concretely, a packaging description is an interface definition in UniCon. A component in UniCon has exactly one interface. Because the component may participate in many forms of interaction, an interface description is consists of an arbitrary number of units, called players, one per form of interaction. Both the interface itself and its constituent players may be annotated with properties (name/value pairs). As an example, here is a component definition in UniCon:

Connector descriptions in UniCon are just as rich as component descriptions. While a component has an interface (divvied into players) and an implementation, a connector has a protocol (divvied into roles) and an implementation. The language constructs apply equally well to connectors and components. However, since connectors are not needed for packaging descriptions, they are not discussed further here.

```
Sort: component
   interface Filter with
      in: player StreamIn with signature: “char*”; end
      out: player StreamOut with signature: “char*”; end
      err: player StreamOut with signature: “char*”; port: StdErr; end
   end
   external Executable with
      executable_file: Filename(“sort”);
   end
end
```

This component's interface is derived from the definition *Filter* and contains three players: *in*, *out*, and *err*. Each of these players has a property *signature*, and the player *err* also has a property *port*.

UniCon is a convenient notation for the system architect's packaging descriptions because of its architectural abstractions. UniCon is also a convenient notation for the packaging specialist because several constructs allow him to record the information that he needs from the system architect: definition inheritance, the *entails* construct, and strongly typed properties.[14]

The packaging specialist captures the information he requires from the system architect in a UniCon interface definition, like the definition *ActiveXControl* in Figure 4.5. In order for a component to have a given packaging, the component's interface must inherit from the packaging specialist's interface definition. For example, for an architect to describe his component as an ActiveX control, his component's interface must inherit from *ActiveXControl*. As with object-oriented languages, when one UniCon definition inherits from another, the sub-definition can be thought to include the textual contents of the super-definition, and a property or player in the sub-definition overrides an item with the same name in the super-definition.

In addition to inheriting contents, a definition may also inherit obligations in the form of *entails* constructs. An *entails* construct names a property or player or set of properties or players that a definition must contain. An *entails* construct has the following form:

entails:	**entails** [**only**] [*range* **of**] *pattern*	
pattern:	*id* : *type type*	property obligation
	player *id*	player obligation
	pattern | *pattern* | … | *pattern*	disjunctive obligation
range:	**exactly** *integer*	
	at least *integer*	
	at most *integer*	
	from *integer* **to** *integer*	

where *id* is a UniCon identifier and *type* is a UniCon type, described later. The default range is *exactly* 1.

An *entails* construct's obligation is fulfilled when the number of properties or players in the definition that match the pattern falls within its range; otherwise, the *entails* construct is unfulfilled. If the *entails* construct is marked *only*, then the pattern must further be exhaustive. If an *entails only* construct has a pattern about players, for example, then all of the definition's players must match the pattern. The *only* clause is used to forbid packaging heterogeneity where the packaging specialist finds it appropriate. These *entails* obligations come due when they are included, directly or though inheritance, in a component or connector definition. Because a component's interface is a leaf in the inheritance tree, it is erroneous for it to contain unfulfilled *entails* constructs. Hence, a definition with unfulfilled *entails* constructs is like an abstract class in an objectoriented programming language, like Java; a component's interface is like a concrete class.

In UniCon, properties are strongly typed and a property's type is considered when matching it against an *entails* construct. The type system includes the following base types, written alongside sample values of that type:

boolean	true, false
integer	-2, -1, 0, 1, 2
range	at least 3, from 1 to 2
real	-2.7, 0.0, 3.1415
string	"hello", "there"
uninterpreted	{can't touch this}

UniCon provides simple functions over these types, such as integer addition and string concatenation. By design, type *uninterpreted* supports no operations whatever. Its function is to allow UniCon to carry the output from external tools without fear that the values will be modified; hence, type *uninterpreted* is a byte-level persistent form to which tools "pickle" their abstract data types. UniCon also supports a parameterized list type and two built-in type constructors, tuples and records:

int list	[1,2,3], []
boolean * integer	(true, 3), (false, 4)
(x: integer, y: real)	(x: 2, y: 4.2), (x: -1, y: 0.0)

UniCon supports user-defined types through a construct like Standard ML's *datatype* construct,[34] restricted to the definition of monomorphic types. When defining a new type, a system architect gives the type a name and describes how to write down values of the new type, possibly using other values. For instance, the declaration

type port_binding = Stdin | Stdout | Stderr | PortNumber **of** int **end**

declares the new type *port_binding*, whose values include *Stdin*, *Stdout*, *Stderr*, and *PortNumber*(5). Using this type system, a packaging specialist can be specific about the kinds of values that a given property can have.

Using UniCon, a packaging specialist spells out the information that the system architect must provide in the packaging description. For example, Figure 4.5 defines *ActiveXControl,* from which the packaging descriptions of all ActiveX controls are derived. The figure illustrates several interesting uses of UniCon. First, many of the definitions contain both a property and an *entails* construct that mentions the same property, which allows the packaging specialist to provide a default value for a required property. Second, type *uninterpreted* is well suited to capture

FIGURE 4.5 The UniCon definition of the ActiveXControl packaging and some of its supporting definitions.

```
type uuid = UUID of uninterpreted end
type com_property_access_type = ReadOnly | ReadWrite end

COM_Property: player BasicPlayer with
    num_assocs: at least 1;
    entails id: type integer;
    entails signature: type string;
    entails access: type com_property_access_type;
    access: ReadWrite;
    entails help_string: type string;
    help_string: "";
    entails value_is_calculated: type boolean;
    value_is_calculated: false;
end

COM_Method: player RoutineDef with
    entails help_string: type string;
    help_string: "";
    entails id: type integer;
end

COM_Interface: player PL_Bundle with
    entails iid: type uuid;
    entails at least 0 of (idl_attribute: type uninterpreted nonaggr);
    entails only at least 0 of (player COM_Property | player COM_Method);
    entails help_string: type string;
    help_string: "";
end

OutgoingInterfaces: player BasicPlayer with
    num_assocs: at least 1;
    entails only at least 1 of (player COM_Interface);
end

COM_Component: interface WindowsDLL with
    entails only at least 1 of (player COM_Interface | player OutgoingInterfaces);
    entails library_iid: type uuid;
    entails coclass_iid: type uuid;
    entails help_string: type string;
    help_string: "";
    entails version: type string;
    version: "1.0";
    entails name: type string;
end

IUnknown: player COM_Interface with
    iid: UUID {00000000-0000-0000-C000-000000000046};

    QueryInterface: player RoutineDef with signature: ([
            "[in] REFIID riid",
            "[out, iid_is(riid)] void **ppvObject"], "HRESULT"); end
    AddRef: player RoutineDef with signature: ([], "ULONG"); end
    Release: player RoutineDef with signature: ([], "ULONG"); end
end
```

```
ActiveXControl: interface COM_Component, WithResources with

    picture_name: "COMinterface";

    iUnknown: player IUnknown;
    iDispatch: player IDispatch;
    iPersistStreamInit: player IPersistStreamInit;
    iQuickActivate: player IQuickActivate;

    resources: player WindowsResources with
        reg: player RegistryResource with
            id: 101;
            registry_instruction_file: Filename("GenericControl.rgs");
        end
        typelib: player TypeLibraryResource with
            id: 1;
            type_library_file: Filename("Generic.tlb");
        end
    end

    dll_defs: player PL_Bundle with
        DllCanUnloadNow: player RoutineDef with
            signature: (["void"], "STDAPI");
            export_publicly: false;
        end
        DllGetClassObject: player RoutineDef with
            signature: (["REFCLSID rclsid", "REFIID riid", "LPVOID* ppv"], "STDAPI");
            export_publicly: false;
        end
        DllRegisterServer: player RoutineDef with
            signature: (["void"], "STDAPI");
            export_publicly: false;
        end
        DllUnregisterServer: player RoutineDef with
            signature: (["void"], "STDAPI");
            export_publicly: false;
        end
    end
    export_from_library: dll_defs;
end


ActiveXControlPackaging: interface ActiveXControl, Packaging with
    AXGen: player PackagingGenerator with
        packager_generator: "DeLineThesis.expert.ActiveXPackagerMaker";
        additional_imports: [ "ActiveX\\ActiveXUtils.uni", "libcmt.uni" ];
        additional_modules: [
            ("imp: COMPONENT ATLImplementationModule;", "imp.all"),
            ("gen: COMPONENT ATLGenericModule;", "gen.all") ,
            ("cmt: COMPONENT MultithreadedLibrary;", "cmt.all")
        ];
        build_option: "/D_MT /I. /I.\\build
                       /I\"\\Thesis\\Software\\VSS\\FlexPack\\Packagings\\ActiveX\"";
    end
end
```

such values as ActiveX's universal identifiers (UUIDs), which may either be canonical (as with IUknown) or may be the output of a tool like Genguid. Third, multiple inheritance is quite handy for defining packagings. The definition of *ActiveXControl*, for example, inherits both from *COM_Component* (because an ActiveX control is a kind of COM component) and *WithResources* (because an ActiveX control must export Windows resources in order to be properly installed). Factoring the definitions in this way allows them to be reused. For example, the definition *WithResources* is also used for defining the packagings for Windows stand-alone applications and Netscape plug-ins.

4.2.2 *Packgen runs the appropriate generator*

Once the system architect writes a packaging description that conforms to the packaging specialist's UniCon definitions, she runs the tool Packgen on the packaging description to invoke the appropriate packaging generator. This tool is invoked with two arguments, a UniCon file and the name of a UniCon interface definition within that file. (If she omits the second argument, the first interface definition in the file is used.) The interface definition on which Packgen is invoked is called the source definition. The source definition must inherit from the following:

```
Packaging: interface BasicInterface with
   entails packager_generator: type java_class;
   entails additional_imports: type string list;
   entails additional_modules: type (string * string) list;
   additional_imports: [];
   additional_modules: [];
   entails at least 0 of (build_option: type string nonaggr);
end
```

Due to this inheritance, the source definition must contain the property *packager_generator*, whose value names a Java class of type PackagerMaker. For example, in Figure 4.5, the definition *ActiveXCon*-trolPackaging inherits both the packaging definition (*ActiveXControl*) and *Packaging*. Hence, to clarify a point from the previous section, an ActiveX control is described in UniCon as a component whose interface inherits directly from *ActiveXControlPackaging* and hence indirectly from *ActiveXControl.*

Given the value of the property *packager_generator,* Packgen creates an instance of the named Java class and invokes one of its methods, as described in the next section. This use of Java's dynamic class loading makes installing new packaging generators very easy. The installation procedure consists of ensuring that the packaging generator's Java class is on the Java path and that the UniCon file that contains the packaging definition is on the UniCon path. Given that these two files are on the right paths, Packgen can find them at run time.

4.2.3 *PackagerMaker captures source code generation*

Reflecting the two steps that the system architect performs to package a component – one step to generate the packager, one step to build the final component – the packaging specialist implements the packaging generator as two Java classes – one that produces the packager, one that produces construction and installation steps. The first class, a subclass of PackagerMaker, is discussed in this section; the second, a subclass of ConfigurationExpert, is discussed in the next.

The first half of a packaging generator is a subclass of the abstract Java class PackagerMaker. As an abstract class, PackagerMaker defers to its subclasses the responsibility for implementing the following method:

```
public abstract CiaoFile[] makePackager(Object intf) throws Die;
```

The argument to this method is an Echo object, described later in Section 4.2.5, that mirrors the source definition on which Packgen is invoked. This method is responsible for producing the set of Ciao/Ciao++ source files that implement the packaging, which are returned as an array of CiaoFile objects. (The generation of non-code artifacts, and construction and installation instructions is discussed in Section 4.2.4.) A CiaoFile object is simply a carrier of aggregate source file information (as might be implemented with a record or structure in another language) and supports only a constructor:

```
public CiaoFile(String filename, int baseLanguage,
                String[] buildOptions, String channelSignature);
```

The arguments are as follows: the name of the source file; the source language (either the constant *BaseLanguage.Ciao* or *BaseLanguage.Ciao-*PlusPlus); any packaging-specific switches to be passed to the compiler (typically the empty string); and a string containing the channel signature for the packager.

This last argument is needed because the packaging generator is responsible for producing a channel signature for each generated source file. The ware provider's tool SigGen cannot be run on a packager's source code. When SigGen is run on a ware's source code, it can readily identify every place where a thread might initially enter the ware, namely those functioned declared as *coroutine*. The places where the thread of control enters the packager, however, is packaging specific. For instance, a thread enters a batch program through the function *main*; a standalone Windows application, through the function *WinMain*; and a procedure library, through any exported procedure. Because of this, the packaging generator itself is responsible for producing the channel signature.

Further, the packaging specialist's generated source code is required to call the function CiaoBegin wherever the thread of control first enters the packager and to call CiaoEnd whenever it last leaves the packager. Knowing the points at which control enters and leaves the component is part of what it means to be the guru for a given packaging. The functions CiaoBegin and CiaoEnd are described in Section 4.3.3.

4.2.4 *Con gurationExpert captures non-source generation*

The previous section discusses how packaging generators produce the source code needed to implement a packaging. This section discusses the generation of non-code artifacts and construction and installation instructions. This split between the two parts of generation reflects the system architect's two-step process for packaging a component: first she generates the packager from the packaging description; then she links the packager with the ware by creating a map between their channel signatures. As previously discussed, to accomplish this first step, she uses Packgen and, through it, a PackagerMaker object. To accomplish the second step, she creates a UniCon description of her final component.

In this UniCon description, like that shown in Figure 4.6, the component's interface is the packaging description from which the packager was generated. The component's implementation is a configuration that consists of two components (the ware and the packager) connected with a Channels connector. This connector has a property named *match* whose value is the channel map. The build the final component, the system architect invokes the UniCon compiler on this description.

When the UniCon compiler builds a component, it invokes a piece of software, called an expert, to generate the non-code artifacts and construction and installation instructions needed to build the component.[48] The compiler looks up the property *expert* on the implementation, which names a Java class that implements the expert. In UniCon, an implementation may either be an external or a configuration. For an external, the expert must be a subclass of the Java class Expert; for a configuration, a subclass of ConfigurationExpert. Subclasses of either class must implement the following methods:

```
public void init(Object intfEcho, Object implEcho);
protected void checkSemantics(Object intfEcho, Object implEcho);
protected HandyVector make(Makefile buildFile, HandyVector dep,
                                        Object intfEcho, Object implEcho);
```

The first method initializes the expert object; the second causes the expert to check whether the implementation is semantically correct, a check that varies based on the nature of the implementation; and the third causes the expert to amend a Makefile with whatever construction and installation steps are needed for this implementation. All three methods are given two Echo objects, one that mirrors the interface definition and one that mirrors the implementation definition. Hence an expert's implementation has the same simplified access to UniCon definitions that PackagerMaker objects do.

A typical implementation of the *make* method uses GlueCode to generate any necessary non-code artifacts. (The Macro tool is inapplicable

```
PNGViewerControl: component
    interface PNGViewerControl_Interface;
    configuration FlexiblePackagingStyle with
        expert: JavaExpert("DeLineThesis.expert.ActiveXExpert");
        ware: component PNG_ViewerWare;
        pkg: component PNGViewerControlPackager;
        anonymous connector Channels with
            connection ware.channels to channels;
            connection pkg.channels to channels;
            match: [
                (pkg.channels.Init, ware.channels.Finalize, TypesOkay),
                (pkg.channels.SetFilename, ware.channels.SetFilename,
                    TypeFixupCode("
                        channel in BSTR WindowsString;
                        channel out char* CString;
                        BSTR bstr;
                        char* cstr = (char*)malloc(100);
                        extern int sprintf(char *buffer, const char *format, ...);
                        in(WindowsString, bstr);
                        sprintf(cstr, \"%S\", bstr);
                        out(CString, cstr);
                    ")),
                (pkg.channels.GetFilename, ware.channels.EchoFilename,
                    TypeFixupCode("
                        channel in char* CString;
                        channel out BSTR WindowsString;
                        char* cstr;
                        BSTR bstr = (BSTR)malloc(200);
                        in(CString, cstr);
                        wsprintfA(bstr, \"%s\", cstr);
                        out(WindowsString, bstr);
                    "))
            ];
        end
    end
end
```

FIGURE 4.6 A flexibly packaged component, described in UniCon. Its interface PNGViewer-Control_Interface is the packaging description from which the packager was generated. The implementation consists of the ware and the packager, connected with a connector whose property "match" is the channel map.

since it only produces source code.) Once the appropriate non-code artifacts have been generated, the expert uses the Makefile object's interface to add construction and installation steps. This interface contains two methods:

```
public void addRule(String fromExtension, String toExtension,
                    String[] commands);
public void addTarget(String target, String[] dependencies,
                    String[] commands);
```

An expert uses the first to add generic rules to the Makefile, for example, a rule that states how to compile a C file to an object file. An expert uses the second to describe how a given list of commands can be used to build a target from a list of dependencies.

In summary, the packaging specialist is responsible for creating three things: a set of UniCon definitions that describe the information that must appear in the system architect's packaging description; a subclass of PackagerMaker that produces source code from the packaging description; and a subclass of ConfigurationExpert that produces the non-code artifacts and construction and installation instructions from the packaging description.

Three tools ease this Java programming task: Echo, described in Section 4.2.5, creates Java objects that mirror UniCon definitions and obviate the need for the packaging specialist to learn about UniCon's internal representations; GlueCode, described in Section 4.2.6, allows a generator to produce source code by inserting strings into a template text file; and Macro, described in Section 4.2.7, allows a generator to produce source code by inserting Ciao AST nodes into a template Ciao AST tree.

Note, however, that compared with ware providers and system architects, there are very few packaging specialists in the world. There are perhaps only several dozen popular ways in which components interact with one another, hence only the need for several dozen packaging specialists. Further, each packaging specialist is a highly trained expert on a given form of component interaction. Because there are few packaging specialists and because these specialists are well trained, there is less motivation for creating good tools for packaging specialists than there is for ware providers and system architects. Nonetheless, in order to carry out the experiments described in Chapter 5, I built nine packaging generator. To lower the effort involved in producing nine generators, I built the tools described in the next three sections. Although these tools are not a significant contribution of Flexible Packaging, they proved to be sufficiently useful in the practice of being a packaging specialist that they are worth mentioning.

4.2.5 *Echo hides UniCon's internal representations*

The packaging generator reads the system architect's packaging definition in order to produce output based on this definition. To do this, the typical approach is to use the UniCon library to parse and check a definition, then to traverse UniCon's abstract syntax tree to read the definition. The disadvantage of this approach is that every packaging specialist must either write his own parser or learn UniCon's internal representations. The Echo tool provides a way around this.

Given a UniCon definition, Echo produces a Java interface that is a definition-specific projection of the AST. For each *entails* construct in the definition, there is a method in the Java interface that may be called to access the items in the definition that match the *entails*. For instance, given the interface definition

```
Filter: interface BasicInterface with
   entails incremental: type boolean;
   entails at least 1 of (player StreamIn);
   entails at least 1 of (player StreamOut);
end
```

Echo produces the following Java interface

```
public interface Filter extends BasicInterface {
   boolean incremental();
   StreamIn[] StreamIn_players();
   StreamOut[] StreamOut_players();
}
```

Generalizing from this example, the *entails* constructs map to Java methods as follows: if the *entails* construct has a property pattern, the method's name is the property's name, and the return type is the property's type; if the *entails* construct has a player pattern, the method's name is the player definition name with a suffix of *_players*, and the return type is the mapping of the player's definition; if the *entails* construct has a role pattern, the method's name is the role definition name with a suffix of *_roles*, and the return type is the mapping of the role's definition; if the *entails* construct has a disjunctive pattern, a separate method is created as above for each disjunct. A UniCon definition is mapped to a Java interface rather than a Java class because Java interfaces support multiple inheritance. For each UniCon definition, Echo also produces a Java class that implements the Java interface and that encapsulates the knowledge about UniCon's AST representation.

For each user-defined type, Echo similarly produces a Java class whose structure reflects the user-defined type. For example, given the definition

```
type port_binding = Stdin | Stdout | Stderr | PortNumber of integer end
```

Echo produces the following Java class

```
public class port_binding
{
    public final static int tagStdin=0, tagStdout=1,
                        tagStderr=2, tagPortNumber=3;
    public int tag;

    public int PortNumber; // available iff tag == tagPortNumber

    public port_binding (AST_Root root, UserDefinedValue uval)
    {
        // implementation omitted
    }
}
```

To read a value of this user-defined type, the generator accesses the fields of a *unix_port_binding* object. The *tag* field allows the generator to test which type constructor (*Stdin*, *Stdout*, *Stderr*, or *PortNumber*) was used to create the value. If a type's constructor has a value associated with it (as is the case for *PortNumber*), then a field named after the type's construct can be used to access that value. For example, if a property *p* has the value *PortNumber*(3) and if *v* is the Java object representing *p*'s value, then *v.tag* == *tagPortNumber* and *v.PortNumber* == 3.

Because Echo creates Java classes that mirror the structure of the UniCon definitions and that hide the details of UniCon's AST representations, the packaging specialist needs only to understand his own definitions and Echo's translation process. As an illustration of the directness that this lends to the source code, consider a Java method that checks whether an ActiveX control's COM interfaces all have unique UUIDs:

```
boolean noDuplicateUUIDs(ActiveXControl ax)
{
    Hashtable seenSoFar = new Hashtable();
    COM_Interface[] interfaces = ax.COM_Interface_players();
    for (int i=0; i<interfaces.size; i++) {
        String id = interfaces[i].iid().UUID;
        if (seenSoFar.contains(id))
            return false;
        else
            seenSoFar.put(id, id);
    }
    return true;
}
```

This method's code directly reflects the definitions in Figure 4.5. Instances of these Echo classes are provided as arguments to *Packager*-

Maker's method *makePackager* and *Con*figurationExpert's methods *init*, *checkSemantics*, and *make*, which were described earlier.

4.2.6 *GlueCode makes substitutions into text templates*

The GlueCode tool allows a packaging specialist to create a text file with named variables and to substitute text for those variables. These variables are of three kinds: scalar variables; list variables; and block variables. Each type of variable is distinguished by its syntax and its substitution behavior. A scalar variable is written as @@<*variable*>@@ and the text substituted for the variable directly replaces the variable in the output file. A list variable is written as @@<<*variable*>>@@. Many pieces of text may be substituted for the same list variable, which causes the line containing the list variable to be repeated once in the output file for each substitution. A block variable is written between two delimiters: @@<begin *block*>@@ and @@<end *block*>@@. Each time the block is instantiated, its text is repeated in the output file. A block typically contains other variables, which have different substitutions for each block instance. Figure 4.7(a) shows an example template file.

I chose the strange bracket syntax to avoid sequences of characters that have meaning in the generated file. A better implementation would allow the user to select the brackets.

The Java classes GlueCode and GlueBlock allow the generator to substitute text for the variables in a template file. Here are the interfaces to these two classes:

```
public class GlueCode
{
   public GlueCode(String templateFile, String filename)
                                          throws IOException;
   public void setVariable(String var, String value);
   public void addToListVariable(String var, String value);
   public GlueBlock newBlock(String var);
}

public class GlueBlock
{
   public void setVariable(String var, String value);
   public void addToListVariable(String var, String value);
   public GlueBlock newBlock(String var);
}
```

The generator creates a new instance of the template file with GlueCode's constructor, whose arguments are the name of the template file and the name of the output file. The method setVariable substitutes a given piece of text for a given scalar variable, the method addToListVariable provides a new piece of text to substitute for a given list variable, and the method newBlock creates a new instance of a given block variable. Because variables can be nested inside blocks, the call to newBlock returns a GlueBlock object that supports the same variable substitution

FIGURE 4.7 An example use of the GlueCode tool. (a) A template file for generating a C function. (b) Java code that instantiates the template file. (c) The instantiate that the Java code produces.

(a)

```
char* @@<<variable>>@@ = “@@<<string>>@@”;

void tempfun()
{
    char* s = @@<strexp>@@
    @@<begin if_stmt>@@
    if (@@<cond>@@)
        printf(“%s”, @@<variable>@@);
    @@<end if_stmt>@@
}
```

(b)

```
GlueCode gluefile = new GlueCode(“template”, “foo.ciao”);
gluefile.addToListVariable(“variable”, “HELLO”);
gluefile.addToListVariable(“string”, “hello”);
gluefile.addToListVariable(“variable”, “THERE”);
gluefile.addToListVariable(“string”, “there”);
gluefile.setVariable(“strexp”, “HELLO”);
GlueBlock block = gluefile.newBlock(“if_stmt”);
block.setVariable(“cond”, “strlen(s) > 0”);
block.setVariable(“variable”, “s”);
gluefile.generate();
```

(c)

```
char* HELLO = “hello”;
char* THERE = “there”;

void tempfun()
{
    char* s = HELLO;

    if (strlen(s) > 0)
        printf(“%s”, s);

}
```

methods. Figure 4.7(b) shows a sample use of these methods, with the results show in Figure 4.7(c).

When GlueCode is used to create Ciao or Ciao++ source code, the generator produces the source code at the lexical level. If the generator, for instance, creates an output file with a syntax error, this error is not discovered until the source code is submitted to the compiler. In contrast, the Macro tool allows the generator to produce source code by making substitutions into an abstract syntax tree, thereby preventing the possibility of syntax errors.

4.2.7 *Macro makes substitutions into abstract syntax trees*

Macro allows a generator to parse Ciao or Ciao++ source code into an abstract syntax tree with named "holes" and to substitute AST nodes into those holes. To create Macro, I extended the grammar of Ciao and Ciao++ to include new hole constructs: the syntax $exp(*ident*) is a valid expression; $stm(*ident*) is a valid statement; and $top(**ident**) is a valid top-level declaration, like a function definition or user-defined type. For all three extensions, the identifier names the hole.

The use of Macro is similar to that of GlueCode. The packaging specialist creates a template file with Ciao or Ciao++ source code, using the hole constructs wherever substitutions should be performed. He then uses the Java class Macro to perform the substitutions:

```
public class Macro
{
    static public C_Expression exp(String s) throws ParseException;
    static public C_StatementList stm(String s) throws ParseException;
    static public C_Root top(String s) throws ParseException;
    static public void subst(C_Root root, String name,
                                        SimpleNode replacement)
}
```

The first three methods parse strings to produce various AST nodes: exp produces an expression; stm produces a statement; and top produces a top-level construct. The method subst takes the root of an AST, the name of a hole, and a AST node to substitute for the named hole. (This is the version of Macro for Ciao; the version for Ciao++ is similar.) The implementation of subst ensures that the right type of AST node is substituted for the named hole. Figure 4.8 shows how Macro would be used to produce the same output as the GlueCode tool in Figure 4.7.

The chief advantage of GlueCode over Macro is its support for lexical substitutions. For example, source code like the following

```
char* s1 = produce_s1();
char* s2 = produce_s2();
char* s3 = produce_s3();
```

FIGURE 4.8 An example use of the Macro tool. (a) A Ciao file with named holes for generating a C function. (b) Java code that instantiates the template AST. (c) The instantiation that the Java code produces. This use of Macro produces the same output as the use of GlueCode in Figure 4.7.

(a)

```
$top(vars)

void tempfun()
{
    char* s = $exp(strexp);
    $stm(if_stmt)
}
```

(b)

```
CiaoParser parser = new CiaoParser();
C_Root ast = parser.parse("template");
Macro.subst(ast, "vars", Macro.top("
    char* HELLO = \"hello\";
    char* THERE = \"there\";
    "));
Macro.subst(ast, "strexp", Macro.exp("HELLO"));
Macro.subst(ast, "if_stmt", Macro.stm("
    if (strlen(s) > 0)
        printf(\"%s\", s);
    "));
ast.print(new PrintWriter(new FileWriter("foo.ciao")));
```

(c)

```
char* HELLO = "hello";
char* THERE = "there";

void tempfun()
{
    char* s = HELLO;

    if (strlen(s) > 0)
        printf("%s", s);

}
```

would be generated with the following GlueCode template

```
char* s@@<<num>>@@ = produce_s@@<<num>>@@();
```

In Macro, the template would consist of an expression like $top(x). The lexical similarity would instead be captured in the source code that manipulates the template:

```
String code = "";
for (int i=1; i<=3; i++)
    code += "char* s" + i + "= produce_s" + i + "();";
Macro.subst(ast, "x", Macro.top(code));
```

The chief advantage of Macro over GlueCode is Macro's support for syntax checking at generation time.

Macro is based on Batory's, Lofaso's, and Smaragdakis's Jakarta Tool Suite (JTS).[2] A programmer uses JTS to create software generators that are both written in Java and that produce Java source code. Because the language in which the generator is written is the same as the generator's target language, the template source code and the generator's source code can be combined, improving clarity. For example, consider the following source code, written in JTS's extension to Java, called Jak:

C_Expression e = **exp**{ 2 < 3 }**exp**;
C_Statement s = **stm**{ if ($e) System.println("hello"); }**stm**;

(The syntax **exp**{*expression*}**exp** and **stm**{*statement*}**stm** are Jak's constructs for expressions and statements, respectively.) The occurrence of $*e* in the declaration of *s* is an escape: the value of the variable *e* is substituted for $*e* when the statement expression is evaluated. Because packaging generators produce Ciao and Ciao++ code (variations on C and C++), I could not directly use JTS. Macro is an attempt to reap the same benefits in a different implementation.

Macro falls short of JTS's benefits in two key ways. First, because the language in which the generator is implemented differs from the language in which the generated code is implemented, JTS's convenient intermixing of generator and generated code is not possible with Macro. Changing Macro to allow the mixing of template source code and generator source code would require creating a super-language that combined the syntaxes of Java and Ciao or Java and Ciao++, which would require considerable effort.

Second, working with C and C++ code at the level of abstract syntax trees is more inconvenient than working with Java at this level due to the infamous "typedef" problem. Syntactic ambiguities in C and C++ make these languages notoriously tedious to parse. For example, the expression

```
x * y;
```

is a variable declaration if x is the name of a type; otherwise it is a useless, but legal multiplication expression. The typical approach to coping with such ambiguities is for the parser to update a symbol table each time it encounters a *typedef* statement; the lexer then uses this symbol table to distinguish type name tokens from identifier tokens. Because of this problem, parsing a simple declaration, like

```
BSTR b;
```

requires the parser to have processed the *typedef* that defines BSTR, which means parsing the header file that contains this type definition. In short, to parse even a short piece of C code into an abstract syntax tree often requires parsing many header files and processing their type definitions. This limitation makes the use of Macro to generate a source file noticeably slower than the equivalent use of GlueCode. This problem does not occur with Java, whose syntax is more civilized, and hence does not effect JTS. As a result of these two limitations with Macro, I more often used GlueCode to produce source code in the packaging generators.

4.2.8 *Packaging generators versus "wizards"*

The use of software generation to simply the task of achieving a given component packaging is not new. An early example is stub generation in RPC systems.[3] More recently, Microsoft provides "wizards" with its Developer Studio environment to ease the task of creating applications with graphical user interfaces and ActiveX controls. With the ActiveX wizard, a programmer answers a series of questions through a dialog box. After he answers the questions, the wizard generates a code template and the necessary non-code artifacts and construction and installation instructions.

Flexible Packaging's software generation provides several advantages over these wizards. First, as a written document, the system architect's packaging description is a permanent part of the system's design record. In contrast, the information typed into a wizard's dialog box is discarded after generation. Second, because the wizard produces a code template, it encourages the very tangling of concerns that Flexible Packaging avoids through its use of channels. As a result, if a wizard user changes her mind about one of the questions, she must generate a new template and copy to the new template whatever modifications she made to the old template. In contrast, with Flexible Packaging, a system architect modifies her packaging description, re-runs Packgen, and modifies the channel map as necessary. Finally, as described in Chapter 6, the design of Flexible Packaging's generators allows the system architect to describe and generator heterogeneous packagings, a capability no wizard currently provides.

As described earlier, the system architect first creates a packaging description in UniCon and runs Packgen on it to produce the packager. She then writes a UniCon description of the final component, whose implementation includes the ware, the packager, and a connector containing the channel map as a property. Finally, she runs the UniCon compiler on the component description to build the final component. As part of building the final component, UniCon invokes the Ciao compiler. Before translating the Ciao (Ciao++) code to C (C++), the Ciao compiler, as pictured in Figure 4.9, uses FDR to ensure that the ware's and packager's channel signatures are compatible. This section describes both the channel signature check and the workings of the Ciao compiler, including the channel run-time implementation.

4.3.1 *Checking the compatibility of channel signatures*

To validate the compatibility of the ware's and packager's channel signatures, the Ciao compiler uses the FDR model checker for CSP.[44] The compiler creates a description of a CSP process that represents the ware and packager executing in parallel and checks whether this parallel process refines a process that participates in an arbitrary number of events and then successfully halts.

Modeling channels To model channels in CSP, the obvious approach is to use CSP's channels, modeling *in*(*C*) with *C*?*x* and *out*(*C*) with *C*!*x*. Unfortunately, this does not have the intended meaning. Consider a ware and a packager that each do an *in* on channel Chan and then successfully terminate:

Ware = in(Chan) → DONE.
Pkg = in(Chan) → DONE.

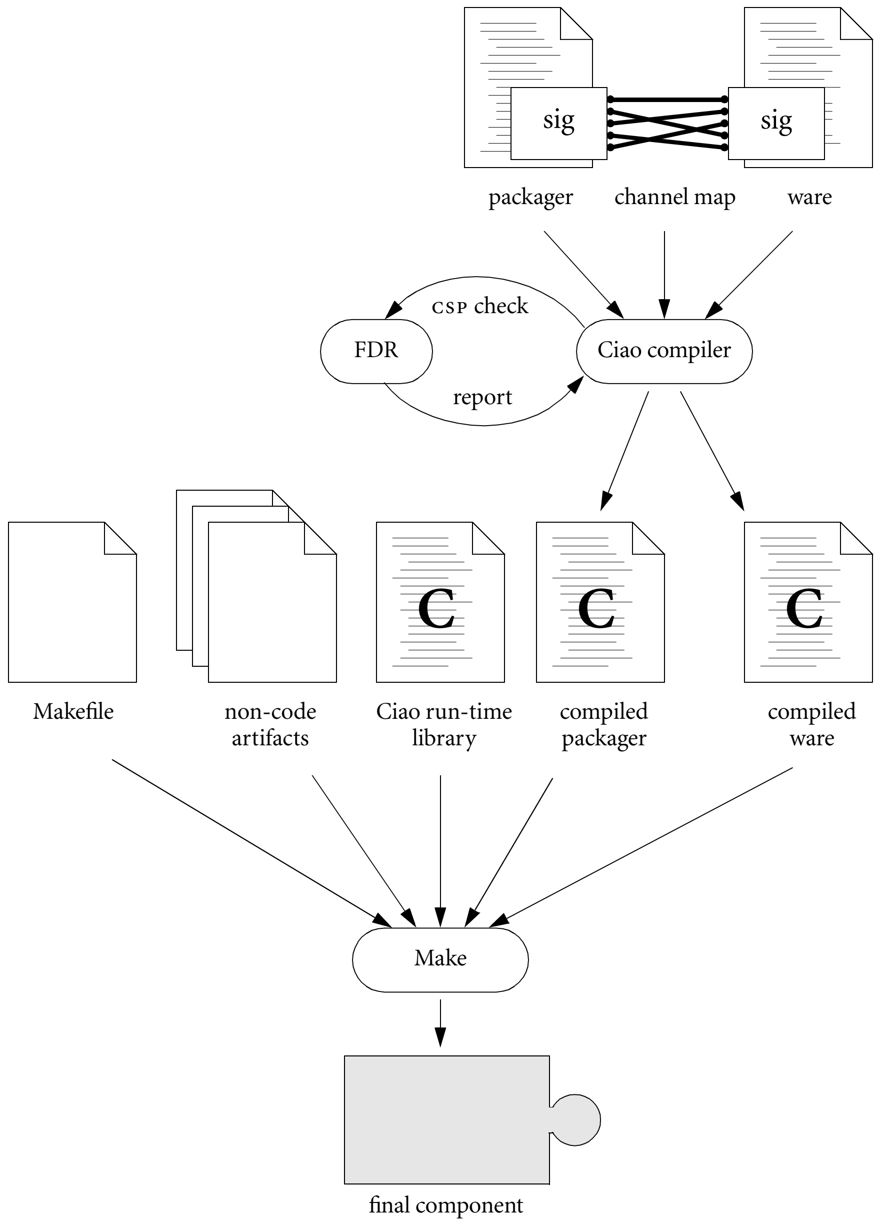
The resulting component would deadlock at run time. However, the proposed CSP model does not deadlock:

Chan?x → SKIP || Chan?x → SKIP

(SKIP is the CSP equivalent of DONE.) In CSP, this process is equivalent to the process Chan?x → SKIP. Hence, when using CSP channels to model Ciao channels, two semantically separate events at the Ciao level (the two *in* statements) are incorrectly unified in the model.

Instead, channel events are modeled with CSP events of the form *C.W.P* or *C.P.W*, where *C* is a channel on which data is flowing from the ware *W* to the packager *P* or vice versa. A ware's event *in*(*C*) is modeled with *C.P.W* (since the data flows from the packager to the ware); its event *out*(*C*) is modeled with *C.W.P*. Hence the previous example would be modeled with the CSP expression

FIGURE 4.9 The system architect's tools. The system architect creates a channel map and runs the Ciao compiler on the map, the ware, and the packager. The compiler first runs FDR to check whether the ware's and packager's channel signatures are compatible and reports FDR's output. If the signatures are compatible, the Ciao compiler translates the Ciao code to C code and runs Make to produce the final component.



Chan.Ware.Pkg → SKIP || Chan.Pkg.Ware → SKIP

which deadlocks since the parallel processes disagree about the first event to occur. If the example were changed to the following deadlock-free component

Ware = in(Chan) → DONE.
Pkg = out(Chan) → DONE.

then the CSP expression

Chan.Ware.Pkg → SKIP || Chan.Ware.Pkg → SKIP

reflects the absence of deadlock.

Modeling in statements When there are exactly two coroutines – one in the ware, one in the packager – we model channels as previously described. However, when there are more than two coroutines, the data that an *in* statement removes from a channel may have been placed there by more than one coroutine. Hence, to model an *in* statement on channel *A* in the channel signature of coroutine *C*, the compiler inspects the channel signatures for the other coroutines, keeping track of which of these channel signatures contains an *out* on *A*. If none of them do, the compiler models the *in* statement with the CSP process STOP, which is the simplest deadlocked process. If exactly one coroutine C_i contains an *out* statement on channel *A*, then the channel is modeled, as previously described, with the event $A.C_i.C$. Finally, if the coroutines $C_1, \ldots, C_k$ each contain at least one *out* on the channel, then the compiler models the *in* statement with the non-deterministic choice $A.C_1.C \rightarrow P \sqcap \ldots \sqcap A.C_k.C \rightarrow P$, where *P* is the CSP process that models the statement following the *in* statement. This use of non-determinism insists that the component not deadlock, no matter which coroutine the run-time scheduler chooses to schedule. In other words, this use of non-determinism ensures that the compatibility of the ware and packager does not depend on any particular coroutine scheduling algorithm.

Modeling out statements The compiler uses CSP's interleave operator (|||) to model the way that *out* statements buffer their output. For instance, the channel signature

Ware = out(A) → in(B) → in(C) → DONE.

is modeled with the CSP expression

(A.Ware.Pkg → SKIP) ||| (B.Pkg.Ware → C.Pkg.Ware → SKIP)

Informally, this CSP expression means that the event *A.Ware.Pkg* may happen before or after any of the events in the expression on the right-hand side of the interleave operator, i.e. before *B.Pkg.Ware*, between *B.Pkg.Ware* and *C.Pkg.Ware*, or after *C.Pkg.Ware*. In short, we model buffering as deferring the execution of an *out* statement until the corresponding *in* statement executes.

(Unfortunately, the use of the interleave operator makes the final CSP check particularly expensive for FDR to verify, because it creates many states for FDR to check. To reduce this expense, the compiler could first model *out* statements with no buffering, using event prefixing ($\rightarrow$) rather than interleaving ($|||$). For example, the previous ware signature would be modeled with the CSP expression

$$\text{A.Ware.Pkg} \rightarrow \text{B.Pkg.Ware} \rightarrow \text{C.Pkg.Ware} \rightarrow \text{SKIP}$$

the first time that the compiler runs FDR. The use of event prefixing greatly reduces the state space to explore, hence speeding the FDR check. If FDR reports that this check fails, then the compiler would re-run the check, this time modeling *out* statements with the interleave operator. The soundness of this optimization relies on a Ciao implementation that implements synchronous versions of *in* and *out* statements until deadlock is detected, in which case it begins buffering *out* statements. Without this restriction, the Ciao implementation is free to choose an order for *in* and *out* statements that is not covered by the optimized FDR check, which allows the possibility of deadlock to go undetected at compile time. The Ciao implementation does not currently uphold this restriction.)

Compatibility check To test for compatibility between the ware's and packager's channel signatures, the compiler uses the CSP refinement test $Spec \sqsubseteq_F System$, where

$$\text{Spec} = (\sqcap\ e : E \bullet e \rightarrow \text{Spec}) \ \sqcap\ \text{SKIP}$$
$$\text{System} = \text{Ware} \parallel \text{Pkg}$$

and E is the set of all events of the form $C.P.W$. The process *Spec* may participate in any number of any of the events in E and then terminate successfully, or it may indefinitely participate in any of the events in E. Informally, *Spec* represents a "good" component – one that either uses channels for a while and then terminates or that uses channels indefinitely (for example, a server component that never quits). In effect, the refinement test checks whether the ware and the package together form such a "good" component.

More formally, the refinement test $Spec \sqsubseteq_F System$ is, by definition, the test $failures(System) \subseteq failures(Spec)$, where $failures(P)$ is the set of all pairs (s, X) such that s is a trace of P and X is the set of events that P can refuse to perform after performing those events in trace s. Given the definition of $\text{SKIP} = \checkmark \rightarrow \text{STOP}$ (where $\checkmark$ is a special event that denotes successful termination) and given the definition of *Spec* above, the set $failures(Spec)$ contains two kinds of elements: a trace that ends in $\checkmark$

paired with all possible subsets of *E* (the process runs to completion and refuses to do anything afterward); or a trace that does not end in ✓ paired with the empty set (the process has not yet successfully terminated and is willing to participate in any event). If *failures*(*System*) is a subset of *failures*(*Spec*), then any element in *failures*(*System*) must also be of one of these two kinds. That is, the process *System* either terminates successfully after some number of events or participates in some number of events and is thereafter willing to participate in any event. Because the events in this refinement test encode *in* and *out* statements, if the refinement test is true, then the composition of the ware and packager either executes *in* and *out* statements until it terminates, or it executes *in* and *out* statements indefinitely; that is, the composition of the ware and packager does not experience run-time deadlock.

When given a refinement check of this kind, the FDR model checker either answers that the refinement is valid or provides a counterexample. The Ciao compiler reports this result to the system architect, translating the CSP events in the counterexample back to the channel operations that the system architect recognizes.

Limitations of the compatibility check This FDR check relies on the accuracy of the ware's and packager's channel signatures. Namely, if the ware or packager can execute a sequence of *in* and *out* statements that is not covered in its channel signature, then there is a possible source of deadlock that the FDR check cannot detect. As described in Section 4.1.3, the accuracy of the channel signatures depends on human programmers, who are fallible. As such, the Ciao library implements run-time deadlock detection. If the FDR check misses a source of deadlock due to inaccuracy in the channel signatures, then the Ciao library can detect it at run time (if it occurs) and provides a log of all *in*, *out*, and *alt* statements for diagnosing the problem.

4.3.2 *Compiling Ciao and Ciao++ to library calls*

Once UniCon establishes the compatibility of the ware's and packager's channel signatures, it invokes the Ciao or Ciao++ compiler on their source code. For each source file, the Ciao compiler first checks that each channel's use is consistent with its declaration: a channel appearing in an *in* statement is declared as an *in* channel; a variable appearing in an *in* statement has the same declared type as the channel; a channel appearing in an *out* statement is declared as an *out* channel; and an expression appearing in an *out* statement is of the same type as the channel's declared type. If the source file passes these checks, the compiler replaces the Ciao constructs with calls to the channel library, whose implementation is described in the next section. Figure 4.10 shows how each Ciao construct is translated into C source code that makes calls to the channel library.

This translation takes the channel map into account. If the channel map shows that a channel has different names in the ware and the pack-

FIGURE 4.10 The implementation of each of the Ciao language constructs as calls on the channel library.

```
channel in type name;                 extern ciao_channel name;
channel out type name;
channel in stream type name;
channel out stream type name;


in(channel, variable);                variable = (type*)CiaoChannelIn(channel);


out(channel, variable);               CiaoChannelOut(channel, & variable);


out(channel, expression);             {
                                        static type temp = expression;
                                        CiaoChannelOut(channel, & temp);
                                      }


alt {                                 {
    in(channel_0, variable_0):          static ciao_channel chs[" + ins.length + "];
        statement_0;                    chs[0] = channel_0;
    in(channel_1, variable_1):          chs[1] = channel_1;
        statement_1;                    ...
    ...                                 chs[k] = channel_k;
    in(channel_k, variable_k):          switch (CiaoAltIn(k+1, chs)) {
        statement_k;                      case 0:
}                                             variable_0 = (type*)CiaoChannelIn(chs[0]);
                                              statement_0;
                                              break;
                                          case 1:
                                              variable_1 = (type*)CiaoChannelIn(chs[1]);
                                              statement_1;
                                              break;
                                          ...
                                          case k:
                                              variable_k = (type*)CiaoChannelIn(chs[k]);
                                              statement_k;
                                              break;
                                        }
                                      }
```

ager, a generated name is substituted for these differing channel names during the translation. During the translation, any conversion code that appears in the channel map is also applied. The conversion code is required to declare exactly one *in* channel and one *out* channel. During the translation, the compiler examines out statements for possible replacement with conversion code. In the channel appearing in an out statement appears in a map entry with conversion code, the out statement is replaced with the conversion. During this replacement, the out statement being replaced is unified with the in statement in the conversion code. For example, given the statements

The current implementation generates the name by concatenating the two differing names in the channel map. If this generated name shadows a name already in use in the source code, the compiler does not catch this mistake.

```
channel out char* Str;
out(Str, "hello");
```

and the channel map entry

```
(Int, Str, TypeFixupCode("
   channel in char* X;
   channel out int Y;
   char* s;
   in(X, s);
   out(Y, strlen(s));"))
```

the compiler will produce the following code

```
channel out char* Str;
{
   char* s;
   s = "hello";             // note the uni  cation of the in and out statements
   out(Int, strlen(s));  // note the channel name replacement
}
```

Allowing the channels in the conversion code to have any name (that is, making the conversion code independent of the channel names in the channel map) allows the system architect to build up a library of conversion routines for use in many maps.

Once the source files have been translated, the compiler's final step is to generate the functions CiaoBegin and CiaoEnd. CiaoBegin contains code to create each of the channels with CiaoMakeChannel and to initialize the coroutine scheduler, discussed in the next section; CiaoEnd contains code to delete each of the channels.

4.3.3 *The Ciao channel library*

The interface to the Ciao channel library is the following:

```
typedef struct ciao_priv_channel_rep *ciao_channel;
typedef void (*ciao_thread_function)(void);

ciao_channel CiaoMakeChannel(char* debugName);
int CiaoChannelEmpty(ciao_channel ch);
void CiaoChannelOut(ciao_channel ch, void* item);
void* CiaoChannelIn(ciao_channel ch);
int CiaoChannelAltIn(int numch, ciao_channel* chs);
void CiaoStreamOpen(ciao_channel ch);
void CiaoStreamClose(ciao_channel ch);
int CiaoStreamMore(ciao_channel ch);
void CiaoCreateThread(ciao_thread_function func);
void CiaoBarrier();
```

Because C does not support polymorphic types, the underlying channel representation is untyped, as witnessed by the *void** argument to CiaoChannelOut and the *void** result from CiaoChannelIn. Hence, the library itself provides no type checking. To provide type checking and to provide a better syntax for *alt* statements than CiaoChannelAltIn provides, Ciao is implemented as a set of language extensions and a compiler. In a language like SML with strong typing, polymorphic types, and higher-order functions, Ciao would be implemented purely as a library, with no compilation step, and would be similar to Reppy's CML.[42] The last two functions in the interface, CiaoCreateThread and CiaoBarrier, support multithreaded packagers and are described in Chapter 6.

A channel is implemented as a dynamic array, initially of length one, which doubles in length each time it becomes full. The coroutining is implemented with the Windows NT Fiber library, which supports non-preemptive, client-scheduled threads. The functions CiaoChannelOut, CiaoChannelIn, CiaoChannelAltIn, and CiaoStreamMore each calls the internal function coroutine_yield to schedule another coroutine (fiber). The function coroutine_yield maintains an ordered list of fibers, with an indication of whether each fiber is blocked on an *in* statement. To schedule a new fiber, coroutine_yield selects the next unblocked fiber on the list. If there are no unblocked fibers to run, coroutine_yield reports deadlock and halts the program. The scheduling scheme is somewhat more complicated in the presence of multithreading, which is discussed in Chapter 6.

5 Experiments

In order to validate the feasibility of the Flexible Packaging method and to demonstrate the thesis claim, I conducted a series of experiments. In the role of the ware developer, I developed three wares that implement three diverse pieces of functionality: image painting; data translation; and text classification. In the role of the packaging specialist, I created nine packaging generators that represent nine packagings used in practice today. Finally, in the role of the system architect, I packaged these wares to form thirteen different components. This chapter catalogs the wares, packagings, and components in the experiments. Chapter 6 discusses how these experiments validate the thesis claim.

5.1 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.

5.1.1 *PNG image painting*

The Portable Network Graphics (PNG) image encoding standard was recently designed to be a successor to the popular GIF standard. One reason that 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, and web browsers; each imposes its own packaging requirements on the image-handling component. Creating a PNG viewing component for each of these niches takes time. This, then, is a natural opportunity for Flexible Packaging. With the functionality of parsing and displaying a PNG image captured as a ware, the functionality can be reused 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.

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
```

(This is the channel signature that SigGen produces, which could be further simplified.) 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 de-allocating its resources.

As the author of this ware, I faced the choice of how flexible to be about ordering. For example, I could have insisted on a more stringent ordering, such as the following:

```
PNG_Main      =  in(Init) → PNG_Main1
PNG_Main1     =  in(SetFilename) →
                    (   out(ErrorMessage) → PNG_Main1
                    [?] in(Paint) → PNG_Main1 )
                 [] in(Finalize) → DONE.
```

This version insists that the packager provide a file name before each paint request and that it send only one paint request per file name. Such ordering guarantees make it somewhat easier to write the ware, for example, by simplifying resource management. However, the penalty is that such a ware cannot be combined with a packager that cannot make the ordering guarantees, like Netscape plug-ins. Instead, as the author, I chose to give up strong ordering guarantees in favor of a wider range of packagers.

5.1.2 *Area code translation*

In order to accommodate an ever increasing need for new telephone numbers in western Pennsylvania, 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). Because of this change, phone numbers had to be updated in many databases and other electronic artifacts. The variety of artifacts is staggering: traditional 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 huge 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.

5.1.3 *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. A Zephyr message, also called a Zephyrgram, consists of the following strings:

- a class, which distinguishes conversational messages (those of class "MESSAGE") from automatically generated notifications;
- an instance, which the message sender chooses to represent the message's subject;
- a sender id, which is the sender's Kerberos authentication ID;
- a sender name, which is text the user chooses to give himself a name;
- a time stamp, which indicates when the message was sent; and
- a body, which contains the message's content.

The Zephyr community uses several programs to read and send messages, each of which display the messages as a list ordered by the message 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:

```
ListenForMessages = in(ZephyrNotices) → Classify; ListenForMessages
                    [] out(ThreaderDone) → DONE.
Classify          =  out(NewMessage) → DONE
                    [?]out(NewThread) → out(NewMessage) → DONE.
```

```
channel in stream (char*, char*, char*, char*, char*, char*) ZephyrNotices
channel out char* NewThread
channel out (char*, char*, char*, char*, char*, char*, char*) NewMessage
```

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. Although ideally a message's body would be used to determine whether it belongs to a given conversation, classifying short text messages from multiple authors is currently beyond today's text classification algorithms. Instead, the ware relies on the Zephyr community's conventions about the message instance strings, namely messages with similar instance strings are part of the same conversation.

5.1.4 *Variety of wares*

This collection of wares represents a variety of ways that functionality can be provided. To characterize these wares, we could consider the space of all wares as having centroids or *pure forms*. We would then classify a ware by saying which pure form it is most like or by saying that it is a hybrid of two or more pure forms. To classify the three experimental wares, consider the following pure forms.

- 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 form:

  ```
  AM  =  in(~) → out(~) → AM
         [] in(~) → out(~) → AM···
         [] in(~) → out(~) → DONE
  ```

- A transducer produces a stream of results from a stream of inputs. It has a channel signature of the following form:

  ```
  Tr  = in(~) → out(~) → Tr [] DONE
  ```

- A driver sends commands out into the world. It has a channel signature of the following form:

  $$\mathrm{Dr} = \mathrm{out}(\sim) \rightarrow \mathrm{Dr}\ [?]\ \mathrm{out}(\sim) \rightarrow \mathrm{Dr}\ [?] \cdots [?]\ \mathrm{out}(\sim) \rightarrow \mathrm{Dr}\ [?]\ \textsc{done}$$

Given this set of pure forms, the PNG ware is an example of an abstract machine, although it differs from the pure form by being more insistent about the order of commands (namely, initialization must happen first). The area code ware is a perfect example of a transducer. The chat message ware can be thought of as a hybrid of a transducer and a driver. Like the transducer, it produces outputs from a stream of inputs, but those outputs are discrete and based on an internal choice, as with a driver.

Since wares are not natural phenomena, but the creations of people, we should not expect a single, naturally arising set of pure forms. Instead, the collection of pure forms is a social convention, based on convenience to those producing and consuming wares. As such, the pure forms above are demonstrative; the actual collection will evolve over time as more experience is gained with Flexible Packaging. Such a collection of pure forms would also be handy for giving guidance about ware/packager compatability. For example, packagings like ActiveX controls, Netscape plug-ins, and Windows applications with GUIs are most likely to be compatable with wares whose pure form is an abstract machine; whereas, Unix filters are most likely to be compatable with transducers. As more experience is gained with Flexible Packaging and a useful collection of pure forms emerges, a packaging specialist will be able to reference these pure forms when he documents his packaging generator in order to give system architects guidance about the wares that are compatable with his packaging.

5.2 EXPERIMENTAL PACKAGINGS

To experiment with a wide variety of packagings that are popular among today's practitioners, I played the role of nine different packaging specialists, 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, I captured enough of the packaging's complexity to be representative without spending more than a few months learning any one technology.

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 the component packagings from the experiments into two categories: *Component Standard* packagings govern how a component's services are invoked; *Data Access* packagings govern how a component accesses external data in order to implement its services. A component's complete packaging, then, consists of *exactly one* Component Standard packaging, plus *zero or more* Data Access packagings. This packaging composition rule allows a component simultaneously to be an ActiveX control, file accessor, and database accessor, but prevents a component from simultaneously being an ActiveX control and a batch program.

This distinction among packagings is reflected in the implementation of the packagers. A packager that implements a Component Standard packaging has a form that the packaging technology dictates. For example, a batch program is implemented as a function called *main*; a Netscape plug-in, as a dynamically loaded library that exports a given set of functions. On the other hand, a packager that implements a Data Access packaging obeys a convention that the Flexible Packaging method introduces. Namely, a Data Access packager is always implemented as a coroutine of the following form:

```
coroutine void DoAccess()
{
    channel in int StartAccess; int ignored; ...

    in(StartAccess, ignored);
    ... do the data access ...
}
```

The *in* statement acts as a guard preventing computation from proceeding until some other computation places data on the channel. The *in* statement may be triggered in two ways. First, as usual, a corresponding *out* statement may appear in the ware's source code. Second, a corresponding *out* statement may appear in another packager's source code, in particular, in the source code of a packager that implements a Component Standard packaging.

To support the latter option, the packaging generators for Component Standard packagings obey a second convention. Because a Component Standard packaging governs how the component's services are invoked, such a packaging determines those points at which control enters the component. Call these the entry points. A batch program, for example, has a single entry point, the function *main*; an ActiveX control has an entry point for each COM method that it exports. A packaging description typically contains one UniCon player for each entry point. The second convention is that an entry point player may have a property called *activate* whose value is a pair: the name of a channel on which to do an *out* statement; and an indication of whether this *out* statement should be executed in a new thread of control. The combination of these two conventions allows packagers to be combined to implement a heterogeneous packaging: each Data Access packager is guarded with an *in* statement; a Component Standard packager may contain the corresponding *out* statement, generated from an *activate* property.

A batch program does not need a player per entry point, since there is exactly one. Its entry point is implicit in its description.

To support the experiments, I created packaging generators for each of the following Component Standard packagings: (1) ActiveX controls;

(2) Netscape plug-ins; (3) Windows applications; (4) batch programs; and (5) CGI scripts. I also created packaging generators to support access to each of the following types of data: (1) Excel spreadsheets; (2) relational databases; (3) text streams; and (4) TCP/IP socket streams. These nine packaging generators are described in the following catalog. An entry in the catalog contains the following information about the packaging: whether the packaging is a Component Standard packaging or a Data Access packaging; the UniCon definitions that a system architect uses to write a packaging description; and an explanation of what the packaging generator does with such a packaging description.

P1 **ActiveX Control** (Component Standard)

UniCon Definitions

```
TYPE uuid = RegistryFormat OF uninterpreted | CppFormat OF uninterpreted END
TYPE com_property_access_type = ReadOnly | ReadWrite END

COM_Property: PLAYER BasicPlayer WITH
    ENTAILS id: TYPE integer;
    ENTAILS signature: TYPE string;
    ENTAILS access: TYPE com_property_access_type;
    access: ReadWrite;
    ENTAILS help_string: TYPE string;
    help_string: "";
    ENTAILS value_is_calculated: TYPE boolean; -- when false, 'get' returns value passed in last 'set'
    value_is_calculated: false;
END

COM_Method: PLAYER RoutineDef WITH
    ENTAILS help_string: TYPE string;
    help_string: "";
    ENTAILS id: TYPE integer;
END

COM_Interface: PLAYER PL_Bundle WITH
    ENTAILS iid: TYPE uuid;
    ENTAILS AT LEAST 0 OF (idl_attribute: TYPE uninterpreted NONAGGR);
    ENTAILS ONLY AT LEAST 0 OF (PLAYER COM_Property | PLAYER COM_Method);
    ENTAILS help_string: TYPE string;
    help_string: "";
END

OutgoingInterfaces: PLAYER BasicPlayer WITH
    ENTAILS ONLY AT LEAST 1 OF (PLAYER COM_Interface);
END

COM_Component: INTERFACE WindowsDLL WITH
    ENTAILS ONLY AT LEAST 1 OF (PLAYER COM_Interface | PLAYER OutgoingInterfaces);
    ENTAILS library_iid: TYPE uuid;
    ENTAILS coclass_iid: TYPE uuid;
    ENTAILS help_string: TYPE string;
    help_string: "";
    ENTAILS version: TYPE string;
    version: "1.0";
    ENTAILS name: TYPE string;
END

-- See page 110 for the definition of WithResources.
```

```
ActiveXControl: INTERFACE COM_Component, WithResources WITH
    iUnknown: PLAYER IUnknown;
    iDispatch: PLAYER IDispatch;
    iProvideClassInfo2: PALYER IProvideClassInfo2;
    iPersistStreamInit: PLAYER IPersistStreamInit;
    iPersistStorage: PLAYER IPersistStorage;
    iQuickActivate: PLAYER IQuickActivate;
    iOleControl: PLAYER IOleControl;
    iOleObject: PLAYER IOleObject;
    iOleInPlaceActiveObject: PLAYER IOleInPlaceActiveObject;
    iViewObjectEx: PLAYER IViewObjectEx;
    iDataObject: PLAYER IDataObject;

    resources: PLAYER WindowsResources WITH
        reg: PLAYER RegistryResource WITH
            id: 101;
            registry_instruction_file: Filename("GenericControl.rgs");
        END
        typelib: PLAYER TypeLibraryResource WITH
            id: 1;
            type_library_file: Filename("Generic.tlb");
        END
    END
END
```

Packaging Generator

The generator uses Microsoft's ActiveX Template Libarary (ATL) to produce the packager's source code. For each of the common COM interfaces, like IViewObject and IOleControl, this library includes a correponding class. To implement a control that exports a given COM interface, a developer creates a class that subclasses from the corresponding ATL class and overrides its methods. The generator produces a class whose superclasses correspond to the COM interfaces listed in the definition *ActiveXControl* above. The generated class has a method for each of the COM methods in each of the COM interfaces above. Since many of these methods are not interesting to the average programmer and since ATL provides reasonable default implementations for them, the generated class overrides only a few, like Draw. The implementation of an overridden method contains an *out* statement to export the method arguments and an *in* statement to import a result (if any) to return. The names of the channels are derived from the method name.

An ActiveX control's packaging description may also introduce new COM interfaces. The generator handles each method in a new COM interface like the overridden methods discussed previously. For each COM property in the new COM interfaces, the generator produces different code, depending the COM property's description in UniCon. In the description, the value of the UniCon property *value_is_calculated* indicates whether the COM property's value is computed or cached. When *value_is_calculated* is true, a computed value is returned when a control does a "get" on the COM property; when *value_is_calculated* is false, the value returned from a "get" is the value from the most recent "set." In the former case, the generator produces two methods, one for the set, one for the get; in the latter case, the generator produces only the set method.

In addition to these methods, the generator produces coroutines. First, it produces a coroutine for error reporting. The coroutine loops reading data from a channel named ErrorMessage and reporting the error messages with a dialog box. If the ActiveX control raises COM events, the generator produces a second coroutine. A COM event is a special kind of COM method. For each COM event *M*, the generator produces a function called Fire_*M*, whose implementation announces *M*. The second coroutine loops over an *alt* statement. For each event *M*, the *alt* statement contains an *in* statement to input the contents of event *M*; each *in* statement guards a call to Fire_*M*.

The generator also produces several non-code artifacts: an IDL file; a registry file; and a resource file. The generator runs the MIDL compiler on the IDL file to produce a type library. The type library advertises the control's capabilities both to developers and to other controls. The registry file is a set of instructions for installing the control in the system registry and is referenced as a resource in the resource file. In addition to this reference to the registry file, the resource file may contain other Windows resources, like an icon to represent the control in development environments like Visual Basic. The generator runs a resource compiler on the resource file. As a final step, it produces the construction steps needed to compile the source code to a dynamically linked library (DLL), to associate the compiled resource file with the DLL through a linker switch, and to run the COM installer to install the DLL.

P2 **Netscape plug-in** (Component Standard)

UniCon Definitions

```
-- See page 110 for the definition of WithResources.

WindowsDLL: INTERFACE BundledLibrary, WithResources WITH
    dynamically_linked: true;
    ENTAILS export_from_library: TYPE player;
END

Netscape4Plugin: INTERFACE WindowsDLL WITH
    ENTAILS mime_type: TYPE string;
    ENTAILS file_extension: TYPE string;
    ENTAILS product_name: TYPE string;
END
```

Packaging Generator

Because the interface to all Netscape plug-ins is the same, the generator always produces the same source file. A Netscape plug-in exports a set of procedure definitions for the browser to call. For each of these procedure, the body contains an *out* statement that outputs the procedure arguments. The generator also produces a coroutine for error reporting, similar to that for ActiveX controls.

In addition to source code, the generator produces a Windows resource file that contains two text properties that the plug-in protocol requires, one for the MIME type and one for the file extension of the documents that the plug-in handles. The generator produces the construction steps needed to compile the source code into a dynamically loaded library (DLL) with the Windows resources and to copy the resulting DLL into the proper directory (a subdirectory named "plugins" of the directory that contains the browser executable).

P3 **Windows application** (Component Standard)

UniCon Definitions

```
Resource: PLAYER BasicPlayer WITH
    num_assocs: AT LEAST 0;
END

VersionInfoResource: PLAYER Resource WITH
    ENTAILS file_version: TYPE uninterpreted;
    ENTAILS product_version: TYPE uninterpreted;
    ENTAILS file_flags_mask: TYPE uninterpreted;
    ENTAILS file_flags: TYPE uninterpreted;
    ENTAILS file_os: TYPE uninterpreted;
    ENTAILS file_type: TYPE uninterpreted;
    ENTAILS file_subtype: TYPE uninterpreted;
    -- these are the defaults the DevStudio wizard gives you
    file_version: {1,0,0,1};
    product_version: {1,0,0,1};
    file_flags_mask: {0x3fL};
    file_flags: {0x0L};
    file_os: {0x4L};
    file_type: {0x2L};
    file_subtype: {0x0L};
    ENTAILS key: TYPE (key_name:string, key_value:string) NONAGGR;
END

BitmapResource: PLAYER Resource WITH
    ENTAILS id: TYPE integer;
    ENTAILS bitmap_file: TYPE file_type;
END

IconResource: PLAYER Resource WITH
    ENTAILS id: TYPE integer;
    ENTAILS icon_file: TYPE file_type;
END

TYPE menu_item_type = MenuItem of (label: string, id:integer) | Separator END

MenuResource: PLAYER Resource WITH
    ENTAILS id: TYPE integer;
    ENTAILS menu_contents: TYPE (string * menu_item_type list) list;
END

RegistryResource: PLAYER Resource WITH
    ENTAILS id: TYPE integer;
    ENTAILS registry_instruction_file: TYPE file_type;
END

TypeLibraryResource: PLAYER Resource WITH
    ENTAILS id: TYPE integer;
    ENTAILS type_library_file: TYPE file_type;
END
```

```
ResourceBundle: PLAYER BasicPlayer WITH
    ENTAILS AT LEAST 1 OF (PLAYER Resource);
    ENTAILS AT LEAST 0 OF (PLAYER VersionInfoResource);
    ENTAILS AT LEAST 0 OF (PLAYER BitmapResource);
    ENTAILS AT LEAST 0 OF (PLAYER IconResource);
    ENTAILS AT LEAST 0 OF (PLAYER MenuResource);
    ENTAILS AT LEAST 0 OF (PLAYER RegistryResource);
    ENTAILS AT LEAST 0 OF (PLAYER TypeLibraryResource);
END

WithResources: INTERFACE BasicInterface WITH
    ENTAILS AT MOST 1 OF (PLAYER ResourceBundle);
END

WindowsApplication: INTERFACE WithResources WITH
    menu_resource: PLAYER WindowsResources WITH
        menu: PLAYER MenuResource WITH
            id: 4;
            menu_contents: [ ("File", [ MenuItem (label:"&Open...", id:40),
                                        MenuItem (label:"&Exit", id:50) ] ) ];
        END
    END
END
```

Packaging Generator

Creating a generator for Windows applications with arbitrary graphical user interfaces (GUIs) would amount to recreating Microsoft's GUI design wizard in Developer's Studio. This wizard allows a designer to create a GUI through a WYSIWYG editor and to generate a source code skeleton that implements the GUI. Rather than recreate this tool, the generator for this packaging creates applications with a fixed graphical user interface. This interface consists of a single window with a menu bar that contains a "File" menu. The "File" menu has two items: "Open," which launches a file-loading dialog box; and "Exit," which terminates the application. The generated application takes a single, optional command-line argument, which is the name of a file to load. Providing this argument is the same as loading the file through the dialog box.

The architect's packaging description must contain an icon Windows resource, which serves as the application's icon on the desktop. Through inheritance, the description also contains a menu resource, shown above. The source code that the generator produces is always the same, except for the resource IDs of the menu and icon. This source code uses *out* channels to indicate when the application is initializing, painting, loading a file, and terminating. The generator also produces a Windows resource file and the construction steps needed to compile the source code to an executable with the given Windows resources.

(To produce a Windows application with a different GUI, a system architect could use Microsoft's GUI designer to generator a code skeleton. The generated source code contains comments that indicate those places where the programmer should insert the code that implements his functionality. The architect would instead replace these comments with channel declarations.)

P4 **Batch program/filter** (Component Standard)

UniCon Definitions

```
ConsoleApplication: INTERFACE BasicInterface WITH
    ENTAILS AT LEAST 0 OF (PLAYER BasicPlayer);
END
```

Packaging Generator

The source code that the generator produces is always the same, namely a definition of the function main:

```
int main(int argc, char* argv[]);
```

The body of main first does an *out* of argv[0] on the channel ProgramName and then an *out* for each of the other elements of the array argv on the stream channel ProgramArguments. Finally, the body of main does an *in* on the channel ProgramResult to input a status code to return as the result of main. The generator also produces the construction steps needed to compile the source code to an executable.

P5 **CGI script** (Component Standard)

UniCon Definitions

```
TYPE cgi_method_type = PostMethod | GetMethod END

CGI_Script: INTERFACE ConsoleApplication WITH
    ENTAILS query_method: TYPE cgi_method_type;
    ENTAILS query_names: TYPE string list;
END
```

Packaging Generator

A CGI script produces a MIME-encoded document (typically an HTML document) based on a query string. There are two kinds of CGI scripts, which differ in how the query string is provided to the script: those that use the "post" query method receive the query string through standard input; those that use the "get" query method receive the query string through command-line arguments. The CGI script packaging generator produces either kind of CGI script, depending on the value of the *query_method* property. A query string is an encoded list of name–value pairs. The source code that the generator produces decodes this string and, for each name in the property *query_names*, looks for a name– value pair with that name and outputs the value on a channel with that name. After outputting the values in the query string, the generated code reads the contents of a channel called HTML output and returns those contents as the result of the script.

The generator also produces the construction steps needed to compile the source code to an executable and to copy the executable to the web server's CGI script directory.

(This is the only generator in the experiments for which channel signatures do not adequately capture the packager's use of channels. For each of the names in *query_names*, the generated packager will output a single value on a channel of that name. These *out* statements can occur in any order. To capture the fact that the *out* statements occur once per channel and in any order is to spell out all the possible orderings in the channel signature. This makes the channel signature tediously long if there are more than a few query names. Instead, the channel signature less accurately states that the *out* statements happen any number of times in any order, which can be succinctly captured with a recursive process definition. In future work, channel signatures could also include CSP's interleave operator (|||) to allow the packaging specialist directly to specify the lack of ordering constraints.)

P6 **Excel spreadsheet** (Data Access)

UniCon Definitions

```
TYPE cell_address_type = Col OF string | Row OF string | FirstEmptyCell | Current END

TYPE cell_access_scope_type =
    CellRange of (cell_address_type * cell_address_type) * (cell_address_type * cell_address_type) |
    Cell of (cell_address_type * cell_address_type)
END

TYPE cell_access_type = CellRead | CellWrite | CellUpdate END

SpreadsheetAccess: PLAYER BasicPlayer WITH
    ENTAILS range: TYPE cell_access_scope_type;
    ENTAILS access: TYPE cell_access_type;
    access: CellUpdate;
    ENTAILS AT LEAST 0 OF (PLAYER SpreadsheetAccess);
END

SpreadsheetRead: PLAYER SpreadsheetAccess WITH
    access: CellRead;
END

SpreadsheetWrite: PLAYER SpreadsheetAccess WITH
    access: CellWrite;
END

SpreadsheetAccessor: INTERFACE Application WITH
    ENTAILS AT LEAST 1 OF (PLAYER SpreadsheetAccess);
END
```

Packaging Generator

The source code that this generator produces uses extensions to Microsoft's C++ compiler that support COM. In addition to the usual *#include* directive, the C++ preprocessor has an *#import* directive which translates COM type libraries (mentioned on page 107) to C++ data structures. This translation maps COM objects to C++ objects, COM method calls to C++ method calls, and COM property accesses to C++ object field accesses. Microsoft Excel is implemented as a collection of COM objects, so invoking Excel, loading spreadsheets, and accessing the content of those spreadsheets can all be accomplished through COM. For each spreadsheet access in the packaging description, the generator produces COM method calls to implement the access. If the access type is CellRead, the generator produces an *out* statement to report the spreadsheet cell's content; if the type is CellWrite, it produces an *in* statement to get a new value for the cell; if the type is CellUpdate, it produces both the *out* and *in* statements.

(Although Microsoft's *#import* directive makes it easy to use COM objects, the *#import* directive itself causes the size of the source code to sky-rocket. In one of the experiments, a source file with a hundred lines of code and a few *#import* directives expands after preprocessing to a source file over four megabytes long and with 22,182 top-level constructs! Because of such large source files, I had to make the Ciao++ compiler an incremental compiler.)

P7 **Relational database** (Data Access)

UniCon Definitions

```
Column: PLAYER BasicPlayer WITH
ENTAILS signature: TYPE string;
    ENTAILS name: TYPE string;
    name: ""; -- this means the player name is the column name
END

Relation: PLAYER BasicPlayer WITH
    ENTAILS AT LEAST 1 OF (PLAYER Column);
END

TYPE odbc_type =  ODBC_Binary | ODBC_Char of integer | ODBC_Date | ODBC_SmallInt |
                  ODBC_Double | ODBC_Float | ODBC_Integer | ODBC_Time END

TYPE sql_statement_type = Select | Update OF (key: (string * odbc_type)) END

DBAccess: PLAYER DBPlayer, OneAssoc WITH
    ENTAILS sql_statement: TYPE sql_statement_type;
    ENTAILS columns: TYPE (string * odbc_type) list; -- name x signature
    ENTAILS from: TYPE string list;
    ENTAILS where: TYPE string;
    where: "";
END

DatabaseAccessor: INTERFACE Application WITH
  ENTAILS AT LEAST 1 OF (PLAYER DBAccess);
    ENTAILS server_name: TYPE string;
END
```

Packaging Generator

This packaging generator uses the popular ODBC library <<CITE?>> to access relational databases. This library allows a program to construct SQL queries and to access the records that result from those queries. For each query (DBAccess player) in the architect's packaging description, the generator produces the necessary calls to the ODBC library. If the SQL statement type is Select, the generated code reports the resulting records through an *out* channel whose name is derived from the DBAccess player; if its type is Update, in addition to reporting the result, it gets a new value for each record through an *in* channel.

P8 **Text stream** (Data Access)

UniCon Definitions

```
TYPE re_format_type =
     EndOfLine |
     Lit OF string |
     InSet OF string |
     NotInSet OF string |
     Choice OF re_format_type list |
     Seq OF re_format_type list |
     Plus OF re_format_type |
     Star OF re_format_type |
     Named OF string * re_format_type
END

TYPE lexeme_set_type = LexemeSet OF string END

TYPE input_format_type = RegularExpression OF re_format_type | Grammar OF lexeme_set_type * string
END

TYPE print_format_type =
     PrintVar OF string |
     PrintS OF string |
     PrintI OF integer |
     PrintR OF real |
     PrintLine OF print_format_type list
END

TYPE output_format_type = Format OF print_format_type | MacroString OF string END

WithFormattedIO: INTERFACE BasicInterface WITH
     InternetLexemes: LexemeSet("internet-lexemes");
END

TYPE unix_port_binding = Stdin | Stdout | Stderr | PortNumber OF integer END

Stream: PLAYER BasicPlayer WITH
   ENTAILS unix_port: TYPE unix_port_binding;
END

StreamIn: PLAYER Stream WITH
   unix_port: Stdin;
     ENTAILS format: TYPE input_format_type;
END

StreamOut: PLAYER Stream WITH
   unix_port: Stdout;
     ENTAILS format: TYPE output_format_type;
END

Filter: INTERFACE WithFormattedIO WITH
   ENTAILS AT LEAST 1 OF (PLAYER StreamIn | PLAYER StreamOut);
END
```

Packaging Generator

For each of the input and output streams in the system architect's packaging description, the generator creates an independent coroutine. For an input stream, this coroutine reads input from the given stream, parses it according to the specified format, and reports the results through *out* channels. For an output stream, the coroutine gathers data through *in* statements and then reports them as specified by the stream's format.

The input stream's format may either be given through a regular expression or through a grammar. If the format is given by a regular expression, the generator creates a Yacc script that parses the regular expression. The treatment of regular expressions is conventional with one exception. Any portion of the regular expression may be given a name (through the Named type constructor). When the Yacc script matches the input against the named portion of the regular expression, the Yacc script reports the matching input on an *out* channel of that name. See page 123 for an example regular expression.

An input stream's format may also be given by a grammar. Rather than having the architect provide a Lex script or other specification of the lexemes (tokens) on which the grammar is based, the generator instead provides a fixed collection of handy lexeme sets. A grammar is then written as a string and uses the lexemes in its chosen lexeme set. (See page 132 for an example grammar.) The generator creates Yacc script for this grammar. The grammars are conventional with one exception. Any non-terminal on the right-hand side of a grammar rule may be enclosed in dollar signs ($). When the Yacc script matches the input against the non-terminal in that context, the Yacc script reports the matching input on an *out* channel named after the non-terminal.

The format of an output stream may be given by print format or a macro string. A print format is a tree of print commands that consists of the following nodes: PrintVar(C) means to print a string from the channel C; PrintS(S) means to print the string S; printI(I) means to print the integer I; PrintR(R) means to print the real R; and PrintLine(L) means to do the commands in L. A macro string is a literal string into which substitutions are made before it is printed. Wherever a string appears between dollar signs ($) in the macro string, the generated code does an *in* statement on a channel whose name is between the dollar signs and substitutes the value from the *in* statement into the macro string. An example of a macro string is on page 123.

P9 **TCP/IP socket stream** (Data Access)

UniCon Definitions

```
-- See page 115 for the definitions of input_format_type and output_format_type.

TYPE request_type = Request of string * output_format_type END
TYPE reply_type = Reply of input_format_type END

TYPE rr_protocol_type = RRProtocol OF (request_type * reply_type) list END

SocketStream: PLAYER BasicPlayer WITH
    ENTAILS hostname: TYPE string;
    ENTAILS port: TYPE integer;
END

RequestReplyProtocolSocketStream: PLAYER SocketStream WITH
    ENTAILS rrprotocol: TYPE rr_protocol_type;
END

SocketClient: INTERFACE BasicInterface WITH
    ENTAILS AT LEAST 1 OF (PLAYER SocketStream);
END

RequestReplyProtocolSocketClient: INTERFACE SocketClient WITH
    ENTAILS AT LEAST 1 OF (PLAYER RequestReplyProtocolSocketStream);
END
```

Packaging Generator

The generator produces source code for a client of an internet server that uses a request–reply protocol over a TCP/IP socket connection. The system architect describes the request–reply protocol using the same input and output format definitions that the text stream packaging uses. (See pages 115 to 116.) The generated source code establishes a socket connection to the specified server and then uses the same generated code as the text stream packaging generator to do input and output on the socket connection.

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). After writing the packaging description, I ran Packgen on it. Packgen, in turn, ran one packaging generator, for a homogeneous packaging, or more than one packaging generator, for a heterogeneous packaging. Either way, Packgen produces a packager, which I then combined with the ware. For most of the components, there were only name and datatype mismatches between the packager and the ware, which I overcame with a channel map. When the mismatch was more complex, I created one or more conversion coroutines to overcome the differences. Finally, when the packager was multithreaded, I created a thread map to assign coroutines to threads.

The following table summarizes the distribution of wares and packagings across the experimental components:

Wares	
PNG image painting	C5, C6, C7
Area code translation	C1, C2, C3, C4
Chat message threading	C8, C9
No ware	C10, C11, C12, C13

Packagings	
ActiveX control	C5, C8, C10
Netscape plug-in	C6
Windows application	C7
Batch program	C3, C9, C12, C13
CGI script	C4, C11
Excel spreadsheet	C1, C12
Relational database	C2, C13
Text stream	C3, C9, C12, C13
TCP/IP socket stream	C8, C9, C10, C11

5.3.1 *Components without wares*

Four of the experimental components are unusual in that they contain no ware. As mentioned in Chapter 2, 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 I designed the packaging generators 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 in-coming message as an ActiveX event. I created this component by writing a packaging description that involves both 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).

5.3.2 *Component testing and performance measurement*

After building each of the experimental components, I tested it by hand. For those components with user interfaces (C1, C2, C3, C7, C9, C12, C13), I ran each component and used the user interface to test it. For the other components, I integrated each component into a system with a user interface and tested the component through the system's user interface. In particular, I tested eachof the CGI scripts (C4 and C11) by creating a web page with a form that the CGI script processes; I tested each of the ActiveX controls (C5, C8, and C10) by building a Visual Basic application that includes an instance of that control; and I tested the PNG Netscape plug-in (C6) by using Netscape to view PNG images.

To ensure that the performance of a flexibly packaged component is comparable to a hand-made component with the same behavior, I created hand-made versions of three of the area code components (components C1, C2, and C3). These three are the only experimental components that both have a ware and that run in batch mode, thereby being suitable for timing. I timed the execution of the three flexibly packaged components and the three hand-made components and compared the two. The run-time overheads for the flexibly packaged components were 8% for the filter (component C3), 2% for the database accessor (component C2), and 1% for the Excel spreadsheet accessor (component C1). These figures are pessimistic for two reasons. First, neither the Ciao compiler nor the channel run-time has been tuned to optimize component performance. Second, the run-time overhead depends on the ratio of the time each module spends computing versus the time the modules spend coordinating through channels. Because the area code ware does little computation, coordination dominates the execution time of the three flexibly packaged components. For these reasons, the run-time overhead, in general, should be less than those measured.

5.3.3 *Reading the catalog*

The catalog that follows describes each experimental component with the following information:

- a name of the component (in bold);
- the ware and the packagings involved;
- an informal description of the component's use;
- the packaging description from which Packgen created the packager;
- the generated packager's channel signature;
- the ware's channel signature;

- the thread map, if any;
- the channel map, if any; and
- the conversion coroutines, if any.

For brevity, the channel signatures in the catalog are not given in their full UniCon syntax, but instead in the abbreviated syntax used in Chapter 1. When a channel signature involves more than one coroutine, it is presented in sections, one section per coroutine. Each section gives the CSP description of the coroutine's channel use and the list of channels that the coroutine uses.

Each catalog entry illustrates the work that I had to perform, in the role of the system architect, to produce the entry's component. First, I had to create the given packaging description. Then I ran Packagen to produce the packager whose channel signature is given. The reader may compare the packaging description with the packager's channel signature to see how the packaging description contributes to the set of channels in the packager, for example, too see how the channel names arise. The reader may also compare the packager's and ware's signatures to understand how I combined them with the given thread map, channel map, and coversion code.

C1

Area code database *Area code translation + batch program–relational database*
A batch program that updates phone numbers in a relational database.
Both the name of the database to update and the SQL query that generates the phone numbers are given in the packaging description.

Packaging Description

```
AreaCodeDatabase: interface DatabaseAccessorPackaging, ConsoleAppPackaging with
    DB: player DBAccess with
        server_name: "NewPhone.dsn";
        sql_statement: Update (key: ("ID", ODBC_Integer));
        columns: [ ("Nighttime Phone", ODBC_Char(50)) ];
        from: [ "Records" ];
    end
    activate: (DB, WithMyThread);
end
```

Packager

```
DBAccess = in(StartDBAccess) → DoDBAccess.
DoDBAccess = out(NighttimePhone) → in(NewNighttimePhone) → DoDBAccess [?] DONE.

    channel in int StartDBAccess
    channel out stream unsigned char* NighttimePhone
    channel in stream unsigned char* NewNighttimePhone

Main = out(ProgramName) → Args.
Args = out(ProgramArguments) → Args [?] out(StartDBAccess)→ DONE.

    channel out char* ProgramName
    channel out stream char* ProgramArguments
    channel out int StartDBAccess
```

Ware

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

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

Channel map

Phone	NighttimePhone	channel in unsigned char* In;
		unsigned char* s1;
		channel out char* Out;
		char* s2;
		in(In, s1);
		s2 = (char*)s1;
		out(Out, s2);

NewPhone	NewNighttimePhone	channel in char* In;
		channel out unsigned char* Out;
		char* s1;
		unsigned char* s2;
		in(In, s1);
		s2 = (unsigned char*)s1;
		out(Out, s2);

C2

Area code spreadsheet *Area code translation + Batch program–Excel spreadsheet*
A batch program that updates cells in a spreadsheet that contain phone numbers. Both the name of the spreadsheet and the range of cells to update are given in the packaging description.

Packaging Description

```
AreaCodeExcelApplicationPackaging: interface SpreadsheetAccessorPackaging, FilterPackaging with
    PhoneNumbers: player SpreadsheetAccess with
        filename: Filename("Records.xls");
        range: CellRange ( (Col "I", Row "2"), (Col "I", FirstEmptyCell) );
    end
    activate: (PhoneNumbers, WithMyThread);
end
```

Packager

```
Spreadsheet = in(StartExcelAccess) → Access0.
Access0 = out(PhoneNumbers) → in(NewPhoneNumbers) → Access0 [?] DONE.

    channel in stream char* NewPhoneNumbers
    channel out stream char* PhoneNumbers

Main = out(ProgramName) → Args.
Args = out(ProgramArguments) → Args [?] out(StartExcelAccess) → DONE.

    channel out char* ProgramName
    channel out char* ProgramArguments
    channel out int StartExcelAccess
```

Ware

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

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

Channel map

NewPhone	PhoneNumbersIn
Phone	PhoneNumbersOut

C3

Area code filter *Area code translation + Batch program–text stream*
A filter that updates phone numbers. The format of the input stream and output stream are given in the packaging description.

Packaging Description

```
AreaCodeFilter: interface FilterPackaging with
    input: player StreamIn with
        format: RegularExpression (
                    Seq [  Named("ID", Plus(InSet("0-9"))), Lit(","),
                           Named("Name", Star(NotInSet(","))), Lit(","),
                           Named("Address1", Star(NotInSet(","))), Lit(","),
                           Named("Address2", Star(NotInSet(","))), Lit(","),
                           Named("City", Star(NotInSet(","))), Lit(","),
                           Named("State", Star(NotInSet(","))), Lit(","),
                           Named("Zip", Star(NotInSet(","))), Lit(","),
                           Named("WorkPhone", Star(NotInSet(","))), Lit(","),
                           Named("HomePhone", Star(NotInSet(","))),
                           EndOfLine ] );
    end
    output: player StreamOut with
        format: MacroString(
                "$IDOut$, $NameOut$, $Address1Out$, $Address2Out$, $CityOut$, $StateOut$, $ZipOut$,
                    $WorkPhoneOut$, $HomePhoneOut$" );
    end
    activate: (input, WithMyThread);
    activate: (output, WithMyThread);
end
```

Packager

```
input = in(Start_input) → input1.
input1 = out(ID) → out(Name) → out(Address1) → out(Address2) → out(City) → out(State) → out(Zip)
            → out(WorkPhone) → out(HomePhone) → input1 [?] DONE.

    channel in int Start_input
    channel out stream char* Address1, Address2, City, HomePhone, ID, Name, State, WorkPhone, Zip

output = in(Start_output) → output1.
output1 = in(IDOut) → in(NameOut) → in(Address1Out) → in(Address2Out) → in(CityOut) →
                in(StateOut) → in(ZipOut) → in(WorkPhoneOut) → in(HomePhoneOut) → output1 []
                DONE.

    channel in int Start_output
    channel in stream char* Address1Out, Address2Out, CityOut, HomePhoneOut, IDOut, NameOut,
                                StateOut, WorkPhoneOut, ZipOut

Main = out(ProgramName) → Args.
Args = out(ProgramArguments) → Args [?] out(Start_output) → out(Start_input) → DONE.

    channel out char* ProgramName
    channel out stream  char* ProgramArguments
    channel out int Start_output
    channel out int Start_input
```

Ware convert = in(Phone) → out(NewPhone) → convert [] DONE.

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

Channel map

Phone	HomePhone
NewPhone	HomePhoneOut
ID	IDOut
Name	NameOut
Address1	Address1Out
Address2	Address2Out
City	CityOut
State	StateOut
Zip	ZipOut
WorkPhone	WorkPhoneOut

C4

Area code CGI script *Area code translation + CGI script*
A CGI script that accepts a phone number to translate from a web form. The web form contains a text box (named "Phone") in which the user enter the phone number. The CGI script generates an HTML document with the updated phone number.

Packaging Description

```
AreaCodeCGIScript: interface CGI_ScriptPackaging with
    query_method: PostMethod;
    query_names: [ "Phone" ];
end
```

Packager

```
Main = out(Phone) → Main [?] GetHTML.
GetHTML = in(HTMLOutput) → GetHTML [?] DONE.

    channel out char* Phone;
    channel in stream char* HTMLOutput;
```

Ware

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

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

Channel map

NewPhone   HTMLOutput

```
channel in char* NewPhone;
channel out stream char* HTMLOutput;
char* s;
static char buffer[200];
in(NewPhone, s);
sprintf(buffer, \"<html><head><title>Zephyr Query Results</
title></head><body>New phone number: %s</body></html>\",
s);
out(HTMLOutput, buffer);
CiaoStreamClose(HTMLOutput);
```

C5

PNG ActiveX control *PNG image painting + ActiveX control*
An ActiveX control that parses and paints PNG images. Setting the control's property named "FileName" causes the control to parse and display the image. The control reports any parse error through a dialog box.

Packaging Description

```
PNGViewerControl: interface ActiveXControlPackaging with
    library_iid: RegistryFormat{7CF18AA0-36FF-11d2-9FD5-00104B33709D};
    coclass_iid: RegistryFormat{7CF18AA1-36FF-11d2-9FD5-00104B33709D};
    name: "PNGViewerControl";
    help_string: "PNG Image Control for DeLine's Thesis";
    IPNGViewerControl: player COM_Interface with
        iid: RegistryFormat{7CF18AA2-36FF-11d2-9FD5-00104B33709D};
        help_string: "PNG Image Interface";
        FileName: player COM_Property with
            id: 1;
            signature: "BSTR";
        end
    end
end
```

Packager

```
Control = out(Init) → Calls.
Calls = out(Paint) → Calls [?] out(SetFileName) → Calls [?] out(Done) → DONE.

    channel out int Init
    channel out struct { struct { long left; long bottom; long top; long right; }* rect; void* hdc; } Paint
    channel out unsigned short* SetFileName
    channel out int Done

ReportErrors    =   in(ErrorMessage) → ReportErrors [] DONE.

    channel in stream char* ErrorMessage
```

Ware

```
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.

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

Channel map	Finalize	Done	

	SetFilename	SetFileName	channel in BSTR windows_string;
			channel out char* c_string;
			BSTR bstr;
			char* cstr = (char*)malloc(100);
			in(windows_string, bstr);
			sprintf(cstr, \"%S\", bstr);
			out(c_string, cstr);

C6

PNG Netscape plug-in *PNG image painting + Netscape plug-in*
A Netscape plug-in that parses and paints PNG images. With the plug-in installed, the browser can load a PNG image as its own document and can display PNG images embedded in HTML documents. As discussed in Chapter 6, this plug-in has the limitation that it can display only a single PNG image at a time. Hence a web page cannot include more than one PNG image.

Packaging description

```
PNGplugin: interface Netscape4Plugin with
    mime_type: "image/png";
    file_extension: "png";
    product_name: "DeLine Thesis PNG Plug-in";
end
```

Packager

```
Plugin = out(New) → Calls.
Calls = out(Paint) → Calls [?] out(NewStream) → Stream [?] out(Destroy) → DONE.
Stream = in(RequestIncremental) → Incr [] in(RequestWholeFile) → out(FileLoaded) → Calls.
Incr   = in(StreamBytesWanted) → out(StreamData) → in(StreamBytesConsumed) → Incr
         [?] out(StreamDone) → Calls
         [?] out(StreamError) → Calls.

    channel out int New, Destroy
    channel out struct { void* hwnd; void* hdc; } Paint
    channel out void* NewStream
    channel out char* FileLoaded
    channel in int RequestIncremental, RequestWholeFile
    channel in int StreamBytesWanted, StreamBytesConsumed
    channel in (int, int, char*) StreamData
    channel in int StreamDone, StreamError

ReportErrors = in(ErrorMessage) → ReportErrors [] DONE.

    channel in char* stream ErrorMessage
```

Ware

```
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.

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

Channel map	Init	New	
	Finalize	Destroy	
	NewFile	FileLoaded	

	Paint	Paint	`channel in struct { void* hwnd; void* hdc; } InPaint;`
			`struct { void* hwnd; void* hdc; } i;`
			`channel out struct { struct { long left, bottom, top, right; }* rect;`
			`void* hdc; } OutPaint;`
			`struct { struct { long left, bottom, top, right; }* rect; void* hdc; } o;`
			`RECT bounds;`
			`in(InPaint, i);`
			`GetClientRect(i.hwnd, (LPRECT)&bounds);`
			`o.rect = &bounds;`
			`o.hdc = i.hdc;`
			`out(OutPaint, o);`

	NewStream	RequestWholeFile	`channel in void* NewStream;`
			`channel out int RequestWholeFile;`
			`void* ignored;`
			`in(NewStream, ignored);`
			`out(RequestWholeFile, 1);`

C7

PNG application *PNG image painting + Windows application*
A stand-alone Windows application that displays PNG images. The application has a single menu "File" with two items, "Open..." and "Exit." The Open menu item displays a dialog box through which the user can load a PNG file. The user may also specify a PNG file on the command line when he launches the application.

Packaging Description

```
PNGWinApp: interface WindowsApplicationPackaging with
    resources: player WindowsResources with
        icon: player IconResource with
            id: 3;
            icon_file: Filename("icon1.ico");
        end
    end
end
```

Packager

```
WinMain = out(Init) → (out(OpenFile) → Cmds [?] Cmds ).
Cmds = out(Paint) → Cmds [?] out(OpenFile) → Cmds [?] out(Done) → DONE.

    channel out int Init
    channel out struct { void* hdc; void* hwnd; } Paint
    channel out char* OpenFile
    channel out int Done

ReportErrors = in(ErrorMessage) → ReportErrors [] DONE.

    channel in char* ErrorMessage
```

Ware

```
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.

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

Channel map	Finalize	Done	
	NewFile	OpenFile	

	Paint	Paint	channel in struct { void* hwnd; void* hdc; } InPaint;
			struct { void* hwnd; void* hdc; } i;
			channel out struct { struct { long left, bottom, top, right; }* rect; void* hdc;
			} OutPaint;
			struct { struct { long left, bottom, top, right; }* rect; void* hdc; } o;
			RECT bounds;
			in(InPaint, i);
			GetClientRect(i.hwnd, (LPRECT)&bounds);
			o.rect = &bounds;
			o.hdc = i.hdc;
			out(OutPaint, o);

C8

Chat message ActiveX control *Chat message threading + ActiveX control–TCP/IP socket stream*
An ActiveX control that listens for Zephyr messages on a socket. When the control receives a message, it decides whether the message is part of an existing conversational thread or a new thread. If it is part of a new thread, it raises an ActiveX event (NewThread) to announce the new thread. It then raises an ActiveX event (NewMessage) to announce a new message within a thread. The control supports two methods: Subscribe, which causes the control to being listening to messages; and Unsubscribe, which causes it to stop listening to messages.

Packaging description

```
ZephyrThreader_Interface: interface SocketActiveXControl with

    iconres: player WindowsResources with
        icon: player IconResource with
            id: 200;
            icon_file: Filename("zephyr.ico");
        end
    end

    ZEchoClient: player RequestReplyProtocolSocketStream with
        hostname: "128.2.198.16"; -- "springer.arch.cs.cmu.edu"
        port: 17763;
        rrprotocol: RRProtocol [
            (Request ("Sub", MacroString("subscribe $SubscriptionClass$ $SubscriptionInstance$\\n")),
                Reply (Grammar (value(InternetLexemes), "
                    Top: (Zgram)* ;
                    Zgram: ClassLine InstanceLine SenderLine TimeLine FingernameLine BodyLine EndLine ;
                    ClassLine: STRING $ZClass$ NEWLINE ;
                    InstanceLine: STRING $ZInstance$ NEWLINE ;
                    SenderLine: STRING $Sender$ NEWLINE ;
                    TimeLine: STRING $Time$ NEWLINE ;
                    FingernameLine: STRING $Fingername$ NEWLINE ;
                    BodyLine: STRING $Body$ NEWLINE ;
                    EndLine: STRING LENGTH_ENCODED_STRING NEWLINE ;
                    ZClass: LENGTH_ENCODED_STRING;
                    ZInstance: LENGTH_ENCODED_STRING;
                    Sender: LENGTH_ENCODED_STRING;
                    Time: LENGTH_ENCODED_STRING;
                    Fingername: LENGTH_ENCODED_STRING;
                    Body: LENGTH_ENCODED_STRING;
                ")))
        ];
    end

    library_iid: RegistryFormat{2502DF00-CC25-11d2-9FEA-00104B33709D};
    coclass_iid: RegistryFormat{2502DF01-CC25-11d2-9FEA-00104B33709D};
    name: "ZephyrThreader";
    help_string: "Zephyr Threading Control for DeLine's Thesis";

    -- (continued on next page)
    IZephyrThreader: player COM_Interface with
        iid: RegistryFormat{2502DF02-CC25-11d2-9FEA-00104B33709D};
        help_string: "The ZephyrThreader control reads Zephyr notices from over a socket and reports them
                with ActiveX events. This control is only necessary because Zephyr/Kerberos are not available
                on WindowsNT.";
        Subscribe: player COM_Method with
            id: 1;
            signature: ( ["BSTR zclass", "BSTR zinstance"], "void");
            help_string: "Begin receiving Zephyr notices of the given class and instance.";
            activate: (ZEchoClient, WithNewThread);
        end
        Unsubscribe: player COM_Method with
            id: 2;
            signature: ( ["void"], "void");
            help_string: "Stop receiving Zephyr notices.";
        end
    end

    Events: player OutgoingInterfaces with
        IZephyrNoticeEvents: player COM_Interface with
            iid: RegistryFormat{3BF4CBC0-CC25-11d2-9FEA-00104B33709D};
            help_string: "This interface is for receiving events about Zephyr notices.";
            OnError: player COM_Method with
                id: 1;
                signature: ( ["BSTR message"], "void" );
                help_string: "Event raised whenever an error occurs.";
            end
            OnNewThread: player COM_Method with
                id: 2;
                signature: ( ["BSTR threadName"], "void" );
                help_string: "Event raised whenever a new Zephyr thread has been identified.";
            end
            OnNewMessage: player COM_Method with
                id: 3;
                signature: ( ["BSTR threadName", "BSTR zclass", "BSTR zinstance", "BSTR sender", "BSTR
                                fingername", "BSTR time", "BSTR body"], "void" );
                help_string: "Event raised whenever a new Zephyr notice (message) arrives.";
            end
        end
    end

end
```

Packager

```
listen_to_socket = in(BeginListeningToSock) → in(Sub) → Sub_do_output; Sub_parse; DONE.
Sub_do_output = in(SubscriptionClass) → in(SubscriptionInstance) → DONE.
Sub_parse = Sub_Top.
Sub_Top = Sub_Top1.
Sub_Top1 = Sub_Zgram; Sub_Top1 [?] DONE.
Sub_Zgram = Sub_ClassLine; Sub_InstanceLine; Sub_SenderLine; Sub_TimeLine;
                    Sub_FingernameLine; Sub_BodyLine; DONE.
Sub_ClassLine = out(Sub_ZClass) → DONE.
Sub_InstanceLine = out(Sub_ZInstance) → DONE.
Sub_SenderLine = out(Sub_Sender) → DONE.
Sub_TimeLine = out(Sub_Time) → DONE.
Sub_FingernameLine = out(Sub_Fingername) → DONE.
Sub_BodyLine = out(Sub_Body) → DONE.

    channel in int BeginListeningToSock
    channel in int Sub
    channel in char* SubscriptionInstance, SubscriptionClass
    channel out char* Sub_ZClass, Sub_Fingername, Sub_Time, Sub_Body, Sub_ZInstance, Sub_Sender

Control = out(Init) → Calls.
Calls = out(Paint) → Calls [?] out(Done) → DONE.

    channel out int Init
    channel out struct { struct { long left; long bottom; long top; long right; }* rect; void* hdc; } Paint
    channel in int Done

ReportErrors = in(ErrorMessage) → ReportErrors [] DONE.

    channel in char* ErrorMessage

ActiveXEventLoop = in(OnError) → ActiveXEventLoop
    [] in(OnNewThread) → ActiveXEventLoop
    [] in(OnNewMessage) → ActiveXEventLoop.

    channel in int FireOnError
    channel in unsigned short* FireOnNewThread
    channel in (unsigned short*, unsigned short*, unsigned short*, unsigned short*, unsigned short*,
                    unsigned short*, unsigned short*) FireOnNewMessage
```

Ware

```
ListenForMessages = in(ZephyrNotices) → Classify; ListenForMessages
    [] out(ThreaderDone) → DONE.
Classify = (out(NewMessage) → DONE [?] Classify)
    [?] out(NewThread) → out(NewMessage) → DONE.

    channel in stream (char*, char*, char*, char*, char*, char*) ZephyrNotices
    channel out int ThreaderDone
    channel out char* NewThread
    channel out (char*, char*, char*, char*, char*, char*, char*) NewMessage
```

Thread map

ListenForMessages	listen_to_socket
ActiveXEventLoop	listen_to_socket
Translate	listen_to_socket

Conversion Coroutine

```
coroutine void Translate()
{
    channel in char* Sub_ZClass;
    channel in char* Sub_ZInstance;
    channel in char* Sub_Sender;
    channel in char* Sub_Time;
    channel in char* Sub_Fingername;
    channel in char* Sub_Body;
    channel out stream (char*, char*, char*, char*, char*, char*) ZephyrNotices;
    char *zclass, *zinst, *sender, *time, *finger, *body;

    while (1) {
        // Gather information from six channels into one
        in(Sub_ZClass, zclass);
        in(Sub_ZInstance, zinst);
        in(Sub_Sender, sender);
        in(Sub_Time, time);
        in(Sub_Fingername, finger);
        in(Sub_Body, body);
        out(ZephyrNotices, zclass, zinst, sender, time, finger, body);
    }
}
```

Channel map	ServerSubscribe	Subscribe	channel in (BSTR, BSTR) Subscribe; channel out int ServerSubscribe channel out char* SubscriptionClass; channel out char* SubscriptionInstance; BSTR zc, zi; static char buf[100], buf2[100]; in(Subscribe, zc, zi); out(ServerSubscribe, 1); sprintf(buf, \"%S\", zc); out(SubscriptionClass, buf); sprintf(buf2, \"%S\", zi); out(SubscriptionInstance, buf2);
	NewThread	FireOnNewThread	channel in char* NewThread; channel out BSTR FireOnNewThread; char* s; int len; wchar_t* wide; BSTR result; in(NewThread, s); len = strlen(s)*2; wide = (wchar_t*)malloc(len); mbstowcs(wide, s, len); result = SysAllocString(wide); out(FireOnNewThread, result);
	NewMessage	FireOnNewMessage	channel in (char*,char*,char*,char*,char*,char*,char*) NewMessage; channel out (BSTR,BSTR,BSTR,BSTR,BSTR,BSTR,BSTR) FireOnNewMessage; char *s1, *s2, *s3, *s4, *s5, *s6, *s7; int len; wchar_t *w1, *w2, *w3, *w4, *w5, *w6, *w7; BSTR r1, r2, r3, r4, r5, r6, r7; in(NewMessage, s1, s2, s3, s4, s5, s6, s7); len = strlen(s1)*2; w1 = (wchar_t*)malloc(len); mbstowcs(w1, s1, len); r1 = SysAllocString(w1); len = strlen(s2)*2; w2 = (wchar_t*)malloc(len); mbstowcs(w2, s2, len); r2 = SysAllocString(w2); len = strlen(s3)*2; w3 = (wchar_t*)malloc(len); mbstowcs(w3, s3, len); r3 = SysAllocString(w3); len = strlen(s4)*2; w4 = (wchar_t*)malloc(len); mbstowcs(w4, s4, len); r4 = SysAllocString(w4); len = strlen(s5)*2; w5 = (wchar_t*)malloc(len); mbstowcs(w5, s5, len); r5 = SysAllocString(w5); len = strlen(s6)*2; w6 = (wchar_t*)malloc(len); mbstowcs(w6, s6, len); r6 = SysAllocString(w6); len = strlen(s7)*2; w7 = (wchar_t*)malloc(len); mbstowcs(w7, s7, len); r7 = SysAllocString(w7); out(FireOnNewMessage, r1, r2, r3, r4, r5, r6, r7);

C9

Chat message digest *Chat message threading + Batch program–text stream–TCP/IP socket stream*
A batch program that breaks a collection of messages from the Zephyr archive into conversations and then prints them in a digest form. The program is intended to allow the user to read a day's worth of Zephyr messages like a newspaper. The batch program takes a single command-line argument, a query to send to the Zephyr archive.

Packaging description

```
ZArchiveClient: interface SocketConsoleApp with
    ZEchoClient: player RequestReplyProtocolSocketStream with
        hostname: "128.2.181.72"; -- "zarchive.cs.cmu.edu"
        port: 19981;
        status_only_response: Grammar(value(InternetLexemes), "
            Top: $Code$ STRING NEWLINE ;
            Code: CODE ;
        ");
        record_response: Grammar (value(InternetLexemes), "
            Top: $Code$ STRING NEWLINE OptRecord ;
            OptRecord: Record | /*nothing*/ ;
            Code: CODE ;
            Record: $Tag$ RecordEnd NEWLINE ;
            RecordEnd: $Value$ | $OpenRec$ (NEWLINE)* (Record)* (NEWLINE)* $CloseRec$ ;
            OpenRec: '[' ;
            CloseRec: ']' ;
            Tag: STRING ;
            Value: LENGTH_ENCODED_STRING ;
        ");
        rrprotocol: RRProtocol [
            (Request ("Start", MacroString("")),
                Reply (value(status_only_response))),
            (Request ("Help", MacroString("HELP\\n")),
                Reply (value(record_response))),
            (Request ("Retrieve", MacroString("RETRIEVE $Query$\\n")),
                Reply (value(record_response)))
        ];
    end
    uses_return_code: true;
    activate: (ZEchoClient, WithMyThread);
end
```

Packager

```
listen_to_socket     =   in(BeginListeningToSock) → (in(Start) → Start_do_output; Start_parse; DONE
                     []  in(Help) → Help_do_output; Help_parse; DONE
                     []  in(Retrieve) → Retrieve_do_output; Retrieve_parse; DONE).
Retrieve_do_output   =   in(Query) → DONE.
Retrieve_parse       =   Retrieve_Top.
Retrieve_Top         =   out(Retrieve_Code) → Retrieve_OptRecord; DONE.
Retrieve_RecordEnd1  =   Retrieve_Record; Retrieve_RecordEnd1 [?] out(Retrieve_CloseRec) → DONE.
Retrieve_RecordEnd   =   out(Retrieve_Value) → DONE [?] out(Retrieve_OpenRec) →
Retrieve_RecordEnd1.
Retrieve_OptRecord   =   Retrieve_Record; DONE [?] DONE.
Retrieve_Record      =   out(Retrieve_Tag) → Retrieve_RecordEnd; DONE.
Help_do_output       =   DONE.
Help_parse           =   Help_Top.
Help_Top             =   out(Help_Code) → Help_OptRecord; DONE.
Help_Record          =   out(Help_Tag) → Help_RecordEnd; DONE.
Help_OptRecord       =   Help_Record; DONE [?] DONE.
Help_RecordEnd1      =   Help_Record; Help_RecordEnd1 [?] out(Help_CloseRec) → DONE.
Help_RecordEnd       =   out(Help_Value) → DONE [?] out(Help_OpenRec) → Help_RecordEnd1.
Start_do_output      =   DONE.
Start_parse          =   Start_Top.
Start_Top            =   out(Start_Code) → DONE.

   channel in int BeginListeningToSock
   channel in int Start, Help, Retrieve
   channel out int Start_Code, Help_Code, Retrieve_Code
   channel out char* Help_CloseRec, Help_Value, Help_OpenRec, Help_Tag
   channel out char* Retrieve_OpenRec, Retrieve_CloseRec, Retrieve_Value, Retrieve_Tag
   channel in char* Query

Main  =  out(ProgramName) → Args.
Args  =  out(ProgramArguments) → Args [?] in(ProgramReturnCode) → DONE.

   channel out char* ProgramName
   channel out stream char* ProgramArguments
   channel in int ProgramReturnCode
```

Ware

```
ListenForMessages = in(ZephyrNotices) → Classify; ListenForMessages
   [] out(ThreaderDone) → DONE.
Classify = (out(NewMessage) → DONE [?] Classify)
   [?] out(NewThread) → out(NewMessage) → DONE.

   channel in stream (char*, char*, char*, char*, char*, char*) ZephyrNotices
   channel out int ThreaderDone
   channel out char* NewThread
   channel out (char*, char*, char*, char*, char*, char*, char*) NewMessage
```

```
char *sender, *instance, *timestring, *fingername;
channel out stream (char*, char*, char*, char*, char*, char*) ZephyrNotices;

int getRecord()
{
    channel in char* Retrieve_Tag;
    channel in char* Retrieve_Value;
    channel in char* Retrieve_OpenRec;
    channel in char* Retrieve_CloseRec;
    char *tag, *ignored, *v, *MESSAGE = "MESSAGE";

    alt {
        in(Retrieve_Tag, tag): {
            alt {
                in(Retrieve_Value, v): {
                    if (strcmp(tag, "sender") == 0) sender = v;
                    else if (strcmp(tag, "signature") == 0)fingername = v;
                    else if (strcmp(tag, "timestring") == 0) timestring = v;
                    else if (strcmp(tag, "instance") == 0) instance = v;
                    else if (strcmp(tag, "body") == 0)
                        out(ZephyrNotices, MESSAGE, instance, sender, timestring, fingername, v);
                }
                in(Retrieve_OpenRec, ignored): { while (getRecord()) ; return 1; }
            }
        }
        in(Retrieve_CloseRec, ignored): return 0;
    }
}

coroutine void StartTranslation()
{
    channel in char* ProgramName;
    channel in stream char* ProgramArguments;
    channel out int Start;
    channel out int Retrieve;
    channel out int BeginListeningToSock;
    channel out int ProgramReturnCode;
    channel out char* Query;
    char* ignored; static char* query;
    extern hash_table ThreadTable;
    extern linked_list ThreadList;

    in(ProgramName, ignored);
    out(BeginListeningToSock, 1);
    out(Start, 1);
    in(ProgramArguments, query);
    out(Retrieve, 1);
    out(Query, query);

    CiaoStreamOpen(ZephyrNotices);
    getRecord();
    CiaoStreamClose(ZephyrNotices);

    print_messages(ThreadList, ThreadTable);
    out(ProgramReturnCode, 0);
}
```

```
coroutine void GatherMessages()
{
    channel in char* NewThread;
    channel in (char*,char*,char*,char*,char*,char*,char*) NewMessage;
    channel in int TheaderDone;
    char *thread, *zclass, *zinst, *sender, *time, *finger, *body;
    int done = 0;
    extern hash_table ThreadTable;
    extern linked_list ThreadList;

    while (!done) {
        alt {
            in(NewThread, thread):
                list_append(ThreadList, thread);
                hash_table_put(ThreadTable, thread, new_list());
            in(NewMessage, thread, zclass, zinst, sender, time, finger, body):
                linked_list list = hash_table_get(ThreadTable, thread);
                message* msg = (message*)malloc(sizeof(message));
                msg->instance = zinst;
                msg->sender = sender;
                msg->time = time;
                msg->fingername = finger;
                msg->body = body;
                list_append(list, msg);
            in(TheaderDone, done):
                done = 1;
        }
    }
}
```

Channel map (none)

C10

Chat message announcer *No ware + ActiveX control–TCP/IP socket stream*
An ActiveX control that receives Zephyr messages from a socket and announces them as ActiveX events. The control provides no functionality; it is just a bridge between a socket connection and ActiveX.

Packaging description

```
ZephyrSocketBridge_Interface: interface SocketActiveXControl with

  ZEchoClient: player RequestReplyProtocolSocketStream with
    hostname: "128.2.198.16"; -- "springer.arch.cs.cmu.edu"
    port: 17763;
    rrprotocol: RRProtocol [
      (Request ("Sub", MacroString("subscribe $ZephyrClass$ $ZephyrInstance$\\n")),
        Reply (Grammar (value(InternetLexemes), "
          Top: (Zgram)* ;
          Zgram: ClassLine InstanceLine SenderLine TimeLine FingernameLine BodyLine EndLine ;
          ClassLine: STRING $ZClass$ NEWLINE ;
          InstanceLine: STRING $ZInstance$ NEWLINE ;
          SenderLine: STRING $Sender$ NEWLINE ;
          TimeLine: STRING $Time$ NEWLINE ;
          FingernameLine: STRING $Fingername$ NEWLINE ;
          BodyLine: STRING $Body$ NEWLINE ;
          EndLine: STRING LENGTH_ENCODED_STRING NEWLINE ;
          ZClass: LENGTH_ENCODED_STRING;
          ZInstance: LENGTH_ENCODED_STRING;
          Sender: LENGTH_ENCODED_STRING;
          Time: LENGTH_ENCODED_STRING;
          Fingername: LENGTH_ENCODED_STRING;
          Body: LENGTH_ENCODED_STRING;
        ")))
    ];
  end

  library_iid: RegistryFormat{719A8400-BAD5-11d2-9FE7-00104B33709D};
  coclass_iid: RegistryFormat{719A8401-BAD5-11d2-9FE7-00104B33709D};
  name: "ZephyrSocketBridge";
  help_string: "Zephyr Socket Bridge for DeLine's Thesis";

  -- (continued on next page)
```

```
    IZephyrSocketBridge: player COM_Interface with
        iid: RegistryFormat{7CF18AA2-36FF-11d2-9FD5-00104B33709D};
        help_string: "The ZephyrSocketBridge control reads Zephyr notices from over a socket and reports
                them with ActiveX events. This control is only necessary because Zephyr/Kerberos are not
                available on WindowsNT.";

        Subscribe: player COM_Method with
            id: 1;
            signature: ( ["BSTR zclass", "BSTR zinstance"], "void");
            help_string: "Begin receiving Zephyr notices of the given class and instance.";
        end

        Quit: player COM_Method with
            id: 2;
            signature: ( ["void"], "void");
            help_string: "Stop receiving Zephyr notices.";
        end

    end

    Events: player OutgoingInterfaces with
        IZephyrNoticeEvents: player COM_Interface with
            iid: RegistryFormat{C12EF3F0-BBB0-11d2-9FE9-00104B33709D};
            help_string: "This interface is for receiving events about Zephyr notices.";
            OnZephyrNotice: player COM_Method with
                id: 1;
                signature: ( ["BSTR zclass", "BSTR zinstance", "BSTR sender", "BSTR fingername",
                                "BSTR time", "BSTR body"], "void" );
                help_string: "Event raised whenever a new Zephyr notice (message) arrives.";
            end
            OnError: player COM_Method with
                id: 2;
                signature: ( ["BSTR message"], "void" );
                help_string: "Event raised whenever an error occurs.";
            end
        end
    end
end
```

Packager

```
listen_to_socket        =  in(BeginListeningToSock) → in(Sub) → Sub_do_output; Sub_parse; DONE.
Sub_parse               =  Sub_Top.
Sub_Top                 =  Sub_Top1.
Sub_Top1                =  Sub_Zgram; Sub_Top1 [?] DONE.
Sub_BodyLine            =  out(Sub_Body) → DONE.
Sub_SenderLine          =  out(Sub_Sender) → DONE.
Sub_TimeLine            =  out(Sub_Time) → DONE.
Sub_FingernameLine      =  out(Sub_Fingername) → DONE.
Sub_ClassLine           =  out(Sub_ZClass) → DONE.
Sub_Zgram               =  Sub_ClassLine; Sub_InstanceLine; Sub_SenderLine; Sub_TimeLine;
                                   Sub_FingernameLine; Sub_BodyLine; DONE.
Sub_InstanceLine        =  out(Sub_ZInstance) → DONE.
Sub_do_output           =  in(ZephyrClass) → in(ZephyrInstance) → DONE.

    channel in int BeginListeningToSock
    channel in int Sub
    channel out char* Sub_Fingername, Sub_ZClass, Sub_ZInstance, Sub_Body, Sub_Sender, Sub_Time
    channel out (unsigned short*, unsigned short*) Subscribe
    channel in stream char* ZephyrInstance, ZephyrClass

Control   =  out(Init) → Calls.
Calls     =  out(Paint) → Calls [?] out(Done) → DONE.

    channel out int Init
    channel out struct { struct { long left; long bottom; long top; long right; }* rect; void* hdc; } Paint
    channel out int Done

ActiveXEventLoop  = in(OnZephyrNotice) → ActiveXEventLoop [ ] in(OnError) → ActiveXEventLoop.

    channel in unsigned short* FireOnError
    channel in (unsigned short*, unsigned short*, unsigned short*, unsigned short*, unsigned short*,
                unsigned short*) FireOnZephyrNotice

ReportErrors = in(ErrorMessage) → ReportErrors.

    channel in char* ErrorMessage
```

Thread map

```
listen_to_socket        ListenForMessage

EventLoop               ListenForMessages
```

Channel map

```
ZephyrClass         ZephyrClass         channel in BSTR windows_string;
                                        channel out char* c_string;
                                        BSTR bstr;
                                        char* cstr = (char*)malloc(100);
                                        in(windows_string, bstr);
                                        sprintf(cstr, \"%S\", bstr);
                                        out(c_string, cstr);

ZephyrInstance      ZephyrInstance      (same as above)
```

C11

Chat message CGI script *No ware + CGI script–TCP/IP socket stream*
A CGI script that takes a Zephyr archive query from a text entry on a web page and produces an HTML document containing the results of the query. This component is a bridge between the CGI script protocol and the Zephyr archive protocol.

Packaging description

```
ZArchiveCGI_Interface: interface CGISocket with
    ZEchoClient: player RequestReplyProtocolSocketStream with
        hostname: "128.2.181.72"; -- "zarchive.cs.cmu.edu"
        port: 19981;
        status_only_response: Grammar(value(InternetLexemes), "
            Top: $Code$ STRING NEWLINE ;
            Code: CODE ;
        ");
        record_response: Grammar (value(InternetLexemes), "
            Top: $Code$ STRING NEWLINE OptRecord ;
            OptRecord: Record | /*nothing*/ ;
            Code: CODE ;
            Record: $Tag$ RecordEnd NEWLINE ;
            RecordEnd: $Value$ | $OpenRec$ (NEWLINE)* (Record)* (NEWLINE)* $CloseRec$ ;
            OpenRec: '[' ;
            CloseRec: ']' ;
            Tag: STRING ;
            Value: LENGTH_ENCODED_STRING ;
        ");
        rrprotocol: RRProtocol [
            (Request ("Start", MacroString("")),
                Reply (value(status_only_response))),
            (Request ("Help", MacroString("HELP\\n")),
                Reply (value(record_response))),
            (Request ("Retrieve", MacroString("RETRIEVE $QueryToRetrieve$\\n")),
                Reply (value(record_response)))
        ];
    end
    activate: (ZEchoClient, WithMyThread);

    query_method: PostMethod;
    query_names: ["Query" ];
end
```

```
Packager    listen_to_socket = in(BeginListeningToSock) →
                          (   in(Start) → Start_do_output; Start_parse; DONE
                              [] in(Help) → Help_do_output; Help_parse; DONE
                              [] in(Retrieve) → Retrieve_do_output; Retrieve_parse; DONE ).
            Retrieve_do_output = in(QueryToRetrieve) → DONE.
            Retrieve_parse = Retrieve_Top.
            Retrieve_Top  = out(Retrieve_Code) → Retrieve_OptRecord; DONE.
            Retrieve_RecordEnd1 = Retrieve_Record; Retrieve_RecordEnd1 [?] out(Retrieve_CloseRec) → DONE.
            Retrieve_RecordEnd = out(Retrieve_Value) → DONE [?] out(Retrieve_OpenRec) → Retrieve_RecordEnd1.
            Retrieve_OptRecord = Retrieve_Record; DONE [?] DONE.
            Retrieve_Record = out(Retrieve_Tag) → Retrieve_RecordEnd; DONE.
            Help_do_output = DONE.
            Help_parse = Help_Top.
            Help_Top = out(Help_Code) → Help_OptRecord; DONE.
            Help_Record = out(Help_Tag) → Help_RecordEnd; DONE.
            Help_OptRecord = Help_Record; DONE [?] DONE.
            Help_RecordEnd1 = Help_Record; Help_RecordEnd1 [?] out(Help_CloseRec) → DONE.
            Help_RecordEnd = out(Help_Value) → DONE [?] out(Help_OpenRec) → Help_RecordEnd1.
            Start_do_output = DONE.
            Start_parse = Start_Top.
            Start_Top = out(Start_Code) → DONE.

               channel in int BeginListeningToSock
               channel in int Start, Help, Retrieve, Done
               channel out char* Help_CloseRec, Help_Value, Help_OpenRec, Help_Tag
               channel out int Help_Code
               channel out int Start_Code
               channel out char* Retrieve_OpenRec, Retrieve_CloseRec, Retrieve_Value, Retrieve_Tag
               channel out int Retrieve_Code
               channel in stream char* QueryToRetrieve

            Main = out(Query) → Main [?] GetHTML.
            GetHTML = in(HTMLOutput) → GetHTML [?] DONE.

               channel out char* Query
               channel out int BeginListeningToSock
               channel in stream char* HTMLOutput

            ReportError = in(ErrorMessage) → DONE.

               channel in char* ErrorMessage
```

```
char *sender, *instance, *timestring, *fingername;
channel out stream char* HTMLOutput;

int getRecord()
{
    channel in char* Retrieve_Tag;
    channel in char* Retrieve_Value;
    channel in char* Retrieve_OpenRec;
    channel in char* Retrieve_CloseRec;
    char *tag, *ignored, *v, *MESSAGE = "MESSAGE";

    alt {
        in(Retrieve_Tag, tag): {
            alt {
                in(Retrieve_Value, v): {
                    if (strcmp(tag, "sender") == 0) sender = v;
                    else if (strcmp(tag, "signature") == 0)fingername = v;
                    else if (strcmp(tag, "timestring") == 0) timestring = v;
                    else if (strcmp(tag, "instance") == 0) instance = v;
                    else if (strcmp(tag, "body") == 0){
                        static char buf[1000];
                        sprintf(buf, "<p><i>%s</i> (<i>%s</i>) (%s) [%s]\r<pre>\r%s</pre>\r\r\r",
                                fingername, sender, instance, timestring, v);
                        out(HTMLOutput, buf);
                    }
                    return 1;
                }
                in(Retrieve_OpenRec, ignored): { while (getRecord()) ; return 1; }
            }
        }
        in(Retrieve_CloseRec, ignored): return 0;
    }
}

static
void die(int c, char* when)
{
    static char buf[100];
    out(HTMLOutput, "<html><head><title>Zephyr Query Results</title></head><body>");
    sprintf(buf, "Unable to contact the zephyr archive (return code %d) %s", c, when);
    out(HTMLOutput, buf);
    out(HTMLOutput, "</body></html>");
}
```

```
coroutine void Fixup()
{
    channel in int Start_Code;
    channel in int Retrieve_Code;
    channel out int Start;
    channel out int Done;
    int c;

    out(Start, 1);
    in(Start_Code, c);
    open_stream(HTMLOutput);
    if (c / 100 != 2) {
        die(c, "on startup");
    } else {
        in(Retrieve_Code, c);
        if (c / 100 != 2) {
            die(c, "after retrieve request");
        } else {
            out(HTMLOutput, "<html><head><title>Zephyr Query Results</title></head><body>\r");
            out(HTMLOutput, "<h1>Zephyr Archive Search Results</h1>\rThe following are the zgrams
                        matching your query:\r");
            getRecord();
            out(HTMLOutput, "</body></html>");
        }
    }
    close_stream(HTMLOutput);
    out(Done, 1);
```

Channel map

Error	ErrorMessage	

Query	QueryToRetrieve	channel in char* QueryIn;
		channel out char* QueryToRetrieveOut;
		channel out int Retrieve;
		char* q;
		in(QueryIn, q);
		out(QueryToRetrieveOut, q);
		out(Retrieve, 1);

C12

Spreadsheet lister *No ware + Batch program–Excel spreadsheet–text stream*
A batch program that dumps cells from a spreadsheet to an output text stream. This component is a bridge between a spreadsheet and a filter.

Packaging description

```
ExcelDumperPackaging: interface SpreadsheetAccessorPackaging, FilterPackaging with
    PhoneNumbers: player SpreadsheetAccess with
        filename: Filename("Records.xls");
        range: CellRange ( (Col "A", Row "1"), (Col "I", FirstEmptyCell) );
        access: CellRead;
    end
    output: player StreamOut with
        format: MacroString("$PhoneNumbers_A$, $PhoneNumbers_B$, $PhoneNumbers_C$,
                             $PhoneNumbers_D$, $PhoneNumbers_E$, $PhoneNumbers_F$,
                             $PhoneNumbers_G$, $PhoneNumbers_H$, $PhoneNumbers_I$");
    end
    activate: (PhoneNumbers, WithMyThread);
    activate: (output, WithMyThread);
end
```

Packager

```
Spreadsheet = Access0.
Access0   =   out(PhoneNumbers_A) → out(PhoneNumbers_B) → out(PhoneNumbers_C) →
                  out(PhoneNumbers_D) → out(PhoneNumbers_E) → out(PhoneNumbers_F) →
                  out(PhoneNumbers_G) → out(PhoneNumbers_H) → out(PhoneNumbers_I) → Access0
              [?] DONE.

    channel out stream char* PhoneNumbers_A, PhoneNumbers_B, PhoneNumbers_C, PhoneNumbers_D,
        PhoneNumbers_E, PhoneNumbers_F, PhoneNumbers_G, PhoneNumbers_H, PhoneNumbers_I
    channel in int StartExcelAccess

output = in(PhoneNumbers_A) → in(PhoneNumbers_B) → in(PhoneNumbers_C) →
             in(PhoneNumbers_D) → in(PhoneNumbers_E) → in(PhoneNumbers_F) →
             in(PhoneNumbers_G) → in(PhoneNumbers_H) → in(PhoneNumbers_I) → output [] DONE.

    channel in stream char* PhoneNumbers_A, PhoneNumbers_B, PhoneNumbers_C, PhoneNumbers_D,
        PhoneNumbers_E, PhoneNumbers_F, PhoneNumbers_G, PhoneNumbers_H, PhoneNumbers_I
    channel in int Start_output

Main = out(ProgramName) → Args.
Args = out(ProgramArguments) → Args [?] DONE.

    channel out char* ProgramName
    channel out char* ProgramArguments
    channel out int Start_output
    channel out int StartExcelAccess
```

Channel map

(none)

C13

Database lister *No ware + Batch program–text stream–relational database*
A batch program that dumps the results of an SQL query on a database to an output text stream. This component is a bridge between a database and a filter.

Packaging description

```
DatabaseDumperPackaging: interface DatabaseAccessorPackaging, FilterPackaging with
    access: player DBAccess with
        server_name: "C:\\Program Files\\Common Files\\ODBC\\DataSources\\NewPhone.dsn";
        sql_statement: Select;
        columns: [ ("Name", ODBC_Char(50)), ("Nighttime Phone", ODBC_Char(50)) ];
        from: [ "Records" ];
    end
    output: player StreamOut with
        format: MacroString("$NameOut$: $PhoneOut$");
    end
    activate: (access, WithMyThread);
    activate: (output, WithMyThread);
end
```

Packager

```
DBAccess = GetName [?] DONE.
GetName = out(Name) → GetNighttimePhone [?] GetNighttimePhone.
GetNighttimePhone = out(NighttimePhone) → DBAccess [?] DBAccess.

    channel out int Start_output
    channel out stream unsigned char* NighttimePhone
    channel in int Start_output
    channel out int StartAccess
    channel in int StartAccess
    channel out stream unsigned char* Name

output = in(NameOut) → in(PhoneOut) → output [] DONE.

    channel in stream char* NameOut
    channel in stream char* PhoneOut

Main = out(ProgramName) → Args.
Args = out(ProgramArguments) → Args [?] DONE.

    channel out char* ProgramName
    channel out stream char* ProgramArguments
```

Channel map

```
Name            NameOut     channel in unsigned char* In; unsigned char* s1;
                            channel out char* Out; char* s2;
                            in(In, s1);
                            s2 = (char*)s1;
                            out(Out, s2);

NighttimePhone  PhoneOut    channel in unsigned char* In; unsigned char* s1;
                            channel out char* Out; char* s2;
                            in(In, s1);
                            s2 = (char*)s1;
                            out(Out, s2);
```

6 Evaluation

The experiments in the previous chapter provide a basis for analyzing the merits of Flexible Packaging. First, I describe how the experiments demonstrate how Flexible Packaging handles several complexities of component packaging: packagings with idiosyncratic construction steps; packagings whose computations are driven by the functionality; packagings that are inherently multithreaded; and heterogeneous packagings. Next, I discuss how Flexible Packaging copes with the unavoidable mismatches that arise when combining wares with packagers. I then use the vocabulary of component interaction from Chapter 2 to describe those aspects of interaction that are encapsulated in the packager and those that are not. Finally, I describe how the experiments validate the thesis claim from Chapter 1.

6.1 HANDLING PACKAGING COMPLEXITY

The packagings that I created for the experiments reflect several complexities that arise in practice. First, several of the packagings require the production of non-code artifacts and the use of packaging-specific construction and installation tools. Second, many of the packagers must handle asynchronous requests from wares. Third, several of the packagers are inherently multithreaded. Finally, most of the packagings are heterogeneous, their packagers the product of several packaging generators.

6.1.1 *Packagers with construction complexities*

For a component to have a given packaging often requires the creation of non-code artifacts and the invocation of packaging-specific construction and installation steps. Several of the experimental packagings involve these construction complexities. The table in Figure 6.1 summarizes these construction complexities for those experimental packagings that have them. The details behind the entries in this table are covered in the packaging catalog in Chapter 5. Were such complexities not to appear in the experiments, one could criticize Flexible Packaging as being inapplicable to many of today's most popular packagings.

6.1.2 *Packagers with additional coroutines*

As discussed in previous chapters, a Ciao module can use an *alt* statement to allow another module to select which of several services to provide. This use of *alt* is handy when the choice among services can be

FIGURE 6.1 Those packagings that involve the creation of non-code artifacts and the invocation of special construction tools and installation steps.

Packaging	*Non-code artifacts*	*Constructions tools*	*Installation steps*
ActiveX control	IDL file registry file resource file	IDL compiler registry compiler linker switch	COM installer
Netscape plug-in	registry file	registry compiler	copy to directory
Windows application	resource file	linker switch	—
CGI script	—	—	copy to directory
Excel spreadsheet	—	compiler construct	—

localized to a particular program location. Often, however, a module is always willing to perform a given service. Rather than tediously using *alt* everywhere, a module can instead use a coroutine dedicated to this service. Several of the packaging generators use this strategy, for example, for error reporting. The packaging generators for ActiveX controls, Netscape plug-ins, and Windows applications each generate a coroutine of the following form:

```
coroutine void ReportErrors()
{
    channel in char* ErrorMessage;
    char* msg;

    while (1) {
        in(ErrorMessage, msg);
        /* report error in packaging-speci  c way */
    }
}
```

Because this computation is in its own coroutine, the packager need not anticipate when the ware will report an error; the packager is always ready to accept an error message.

The ActiveX packaging generator uses the same strategy for announcing events. The generator outputs a coroutine like that above with the following statements inside the *while* loop:

```
alt {
    in($Event_1$, $args_1$): announce $Event_1$ with $args_1$
    in($Event_2$, $args_2$): announce $Event_2$ with $args_2$
    ...
    in($Event_k$, $args_k$): announce $Event_k$ with $args_k$
}
```

The coroutine allows the ware to announce an event by sending data to the channel corresponding to that event.

The final example of this strategy is the text stream packaging generator. For each input stream, the generator produces a separate coroutine that loops reading from the input stream and reporting the resulting data on an *out* channel. For each output stream, the generator produces a separate coroutine that loops reading from an *in* channel and reporting the result on the output stream. This allows the ware to produce and consume data to and from the streams in whatever order it will. This use of coroutines represents a well known solution to an old problem: the problem of structure clash.[27] A classic example is the problem of copying an input file with 80 characters per line to an output file with 120 characters per line. A solution in a typical imperative language cannot be implemented with two straightforward loops, one from one to 80, one from one to 120; one of the two loops must be folded into the structure of the other. With coroutines, each of the two loops can be placed in its own coroutine and retain its straightforward structure. The text stream packaging generator uses this same approach on a similar problem.

In the simple case where there are two coroutines – one in the ware, one in the packager – the thread of control simply passes back and forth between them. Scheduling the thread among the coroutines is simple: when the current module executes an *in* or *out* statement, the thread resumes in the other module. When there are more than two coroutines, however, thread scheduling is more complicated: when the current module executes an *in* or *out* statement, there is more than one "other" module in which the thread could resume. Furthermore, scheduling the wrong coroutine can lead to incorrect component behavior.

Consider a ware that is packaged as an RPC server that reports errors to a log file, shown in Figure 6.1(a). The thread of control begins, as always, in the packager, in this case with a remote procedure call to SetAge. When the thread reaches the *out* statement on Age, the thread may either resume the coroutine Ware or the coroutine ReportErrors. If ReportErrors is selected, the thread would become blocked on the *in* statement on Error. If Ware is selected, execution proceeds until the *out* statement on Error. At this point, there is an interesting scheduling choice. If ReportErrors is scheduled, the error is reported as expected. However, if SetAge is scheduled instead, the thread of control is returned to the RPC caller, without the error being reported.

If the procedure SetAge had a return value, it would contain an *in* statement before the *return* in order to get the value to return to the caller. This extra *in* statement would prevent the bad behavior just dis-

FIGURE 6.1 (a) A component that manifests a potential scheduling problem. (b) An overly restrictive solution to the problem. (c) The use of a barrier to solve the problem.

(a)

```
coroutine void Ware()
{
    channel in int Age;
    channel out char* Error;
    int i;
    in (Age, i);
    if (i < 0)
        out(Error, "bad age");
}
```

```
RPC_STATUS SetAge(int a)
{
    channel out Age;
    out(Age, a);
    return RPC_OKAY;
}

coroutine void ReportErrors()
{
    channel in char* Error;
    char* msg;
    while (1) {
        in(Error, msg);
        fprintf(LOGFILE, "%s\n", msg);
    }
}
```

(b)

```
coroutine void Ware()
{
    channel in int Age;
    channel in int Dummy;
    channel out char* Error;
    int i;
    in (Age, i);
    if (i < 0)
        out(Error, "bad age");
    out(Dummy, 1);
}
```

```
RPC_STATUS SetAge(int a)
{
    channel out Age;
    channel in Dummy;
    int ignored;
    out(Age, a);
    in(DummyReturn, ignored);
    return RPC_OKAY;
}

coroutine void ReportErrors()
{
    channel in char* Error;
    char* msg;
    while (1) {
        in(Error, msg);
        fprintf(LOGFILE, "%s\n", msg);
    }
}
```

(c)

```
coroutine void Ware()
{
    channel in int Age;
    channel in int Dummy;
    channel out char* Error;
    int i;
    in (Age, i);
    if (i < 0)
        out(Error, "bad age");
}
```

```
RPC_STATUS SetAge(int a)
{
    channel out Age;
    out(Age, a);
    CiaoBarrier();
    return RPC_OKAY;
}

coroutine void ReportErrors()
{
    channel in char* Error;
    char* msg;
    while (1) {
        in(Error, msg);
        fprintf(LOGFILE, "%s\n", msg);
    }
}
```

cussed. Given this, one solution to the scheduling problem, shown in Figure 6.1(b), is for SetAge to contain an *in* statement (conveying no useful value) before the *return* and for Ware to contain the corresponding *out* statement. Although this ensures that ReportErrors is scheduled before SetAge returns, this approach imposes more of the packager's control structure onto the ware, making the ware less reusable.

Instead, the Flexible Packaging method uses a barrier construct. In Figure 6.1(c), SetAge contains a call to CiaoBarrier before the *return* statement. A call to CiaoBarrier blocks until all other coroutines are blocked on *in* statements or calls to CiaoBarrier. In this case, because ReportErrors is ready to run, the call to CiaoBarrier would block until ReportErrors is run, reports its error, and gets stuck on the next *in* statement. To ensure that the component's computation has made as much progress as it can, a packager makes a call to CiaoBarrier at every program point just before the thread to control leaves the component. Each packaging generator is responsible for ensuring that the packagers that it produces make the appropriate calls to CiaoBarrier. This barrier solution prevents the unwanted thread scheduling without reducing the ability to mix-and-match packagers and wares.

6.1.3 *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. Were the called control 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.

Multiple threads for multiple control loops This use of multiple threads of control is one solution to Garlan, Allen, and Ockerbloom's multiple control loop problem.[19] 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 in-coming remote procedure calls; a loop to listen for in-coming 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 author.

Combining coroutines and threads Care must be taken when implementing coroutines in the face of multiple threads of control. For Flexible Packaging, a coroutine is thought to "belong" to a thread: a thread starts the execution of a coroutine, and the coroutine always executes within that thread. This coroutine ownership model raises the question of how to assign coroutines to threads.

Deciding whether two coroutines should execute within the same thread is not a question of their sharing channels but a question of whether their computations need to run together. Returning to the chat message ActiveX control (component C8), there are two threads: a thread that enters the control when one of its COM methods is called (represented by *Control* in the channel signature); and a thread that sits and listens to the socket (represented by *listen_to_socket*). There are three other coroutines: *ReportErrors* in the packager; *ActiveXEventLoop* in the packager; and *ListenForMessages* in the ware. Given the system architect's intentions for this component, the coroutine *ListenForMessages* should be run with *listen_to_socket*, since the former processes the Zephyr messages that the latter produces. Similarly, the coroutine *ActiveXEventLoop* should be run with *ListenForMessages*, since the latter uses the former to announce new conversations and messages within conversations. Running either *ListenForMessages* or *ActiveXEventLoop* with *Control* would be a mistake since the control would then process and announce Zephyr messages only when a COM method call is made. Since the assignment of coroutines to threads requires human judgment about the component's computational goals, the system architect provides a declarative thread map to describe the desired associations.

6.1.4 *Packagers from multiple generators*

As discussed in the previous chapter, I split the packagings into two categories, Component Standard packagings and Data Access packagings. I created conventions for the two categories of packagers to allow packagers to be composed to implement heterogeneous packagings. The composition rules is that a component's complete packaging must consist of exactly one Component Standard packaging, plus zero or more Data

	ActiveX	*Netscape*	*Win app*	*Batch*	*CGI*
(*none*)	C5	C6	C7		C4
Excel				C1	
Database				C2	
Text stream				C3	
Socket	C8, C10				C11
Excel + Text				C12	
Database + Text				C13	
Socket + Text				C9	

FIGURE 6.2 The combinations of Component Standard and Data Access packagings that appear in the experiments. All of the combinations are plausible. The regularity of the table entries is an accidental artifact of the experiments and is not significant.

Access packagings. Any combination of packagings that upholds this composition rule is a plausible component packaging. Of these possible combinations, those that the experiments exercised are shown in Figure 6.2, where the Component Standard packagings are in the columns and combinations of the Data Access packagings are in the rows. The regularity apparent in this table is an accidental artifact of the components that I chose to build for the experiments. Any entry in the table above is plausible because of the rule of packaging composition.

6.2 HANDLING MODULES FROM MULTIPLE AUTHORS

In each experiment, I created either a channel map, additional coroutines, or both to combine the packager and the ware. The channel map overcomes differences in channel names and types; the coroutines overcome any other difference. Channel maps alone were sufficient to combine the packager and the ware in all but three experiments.

6.2.1 *Using channel maps*

Most of the experiments used channel maps to match channels with different names or datatypes. These uses of channel maps demonstrate three points. First, the three PNG components (components C5, C6, and

C7) point out that channel maps are unavoidable when a ware is combined with multiple generated packagers.

Consider how the PNG packagers provides the ware with the name of a PNG file. For the ActiveX control, this file name comes from a COM property named *FileName*. The ActiveX packaging generator uses the property name as the basis for the name of the channel that reports that property: *SetFileName*. For the Netscape plug-in, loading a file is event-oriented due to the network; the browser either incrementally loads the file and announces the arrival of more data or it loads the file as a whole and announces that the whole file has been loaded. Hence the Netscape plug-in packaging generator chooses the name *FileLoaded* for its channel. For the Windows application, the file is loaded through the "File/Open" menu item, hence its packaging generator's choice of *OpenFile* as the channel name. Each of these three channel names makes sense in its own context but differs from the others; even if a ware author contrived to use one of these names for his own channel, the ware would not match either of the other two. This name diversity among the packaging generators is not gratuitous, but reflects the different sources of the file information: a property versus the network versus a dialog box.

Similarly, different data representations are native to different kinds of interaction. Filters, for example, represent text in ASCII; whereas, ActiveX controls represent text in Unicode. A ware author who chooses his text representation to be compatible with a filter packager will cause the ware to be incompatible with an ActiveX packager. In short, the natural diversity of names and data representations among packagings makes the use of channel maps unavoidable.

The second point is that a channel map need not connect ware channels to packager channels, but may connect channels from different coroutines in the packager. For example, with the area code filter (component C3), data on the input stream that is to remain unchanged is mapped to corresponding data on the output stream. As a second example, with the chat message ActiveX control (component C8), the arguments to the Subscribe COM method are mapped to the SvrSub socket command. This example also illustrates mapping a single channel with compound information to several channels with simpler information.

Finally, a channel map can map a scalar channel to a stream channel, as the area code CGI script (component C4) illustrates. The compiler notices the difference in arity and inserts calls to CiaoStreamOpen and CiaoStreamClose where needed.

6.2.2 *Using additional coroutines*

In three of the experiments, I added additional coroutines to the component when combining the packager and the ware. First, such coroutines are useful for overcoming mismatches between the ware's and packager's channels when the mismatches are too complicated to overcome with the channel map. For instance, with the chat message digest (component C9), I needed to translate between the packager channels that provide information from the Zephyr archive (Tag, Value, OpenRec, and CloseRec) and the ware channel that consumes Zephyr messages (ZephyrNotices). This translation requires parsing information from the packager's channels and aggregating the information for the ware's channel. The information on the packager's channel is available according to the following protocol:

Record = out(Tag) → RecordEnd.
RecordEnd = out(Value) → DONE [?] out(OpenRec) → RecordEnd1
RecordEnd1 = Record; RecordEnd1 [?] out(CloseRec) → DONE.

These tags and values (like tag "sender", value "rdeline@cs") constituent pieces of a Zephyr message. The translation coroutine records these tags and values to build up a message to send on the ware's channel ZephyrNotices. The chat message CGI script (component C11) uses a coroutine for a similar translation task.

When the system architect wants to supplement the functionality that a ware provides, the typical approach is to package the ware into a component and to compose that component with other components. Another approach, useful when the additional functionality is meager, is to supplement the ware with coroutines that implement the additional functionality. The chat message digest (component C9) illustrates this. The ware provides information about the conversation threads through two channels, one that announces new threads, one that announces new messages on a thread. The order in which the ware produces output on the two channels reflects the order in which it receives Zephyr messages, namely in time order. Rather than showing the messages sorted by time, the digest bundles them by conversation thread. To accomplish this, I added a coroutine, called GatherMessages (page 140), to store the message announcements into a table and then to display the messages by thread.

6.3 FLEXIBLE PACKAGING, ASPECT BY ASPECT

One way to evaluate the Flexible Packaging method is to use the vocabulary of component interaction introduced in Chapter 2. From the point of view of Flexible Packaging, reusing a component in a new system means combining its ware with a new packager. The more the packager insulates the ware from system-specific interaction dependencies, the more reusable the ware. To judge how well Flexible Packaging supports component reuse, then, we could ask the following question: for each aspect of interaction, how well does the packager insulate the ware from a change in that aspect? Given this question, the main contribution of Flexible Packaging is its use of coroutines to insulate the ware from differences in the Data and Control Transfer aspect of interaction. Flexible Packaging also insulates the ware from changes in the Connection Establishment and Data Representation aspects. Currently, however, Flexible

Packaging does not insulate the ware from changes in the Failure, State Persistence, and State Scope aspects.

6.3.1 *Data and Control Transfer*

As mentioned in Chapter 3, a change in the Data and Control Transfer aspect typically has a large impact on a traditionally developed component. A change in this aspect, for example, can affect the overall structure of the code, like whether it is structured using internal control (e.g. file-based interaction) or external control (e.g. event-based interaction). In contrast to traditional development, the use of coroutines between the packager and the ware allows these modules to have independent control structures. A packager that accesses a file, for instance, can be replaced with a packager that interacts through messages without changing the ware, as was illustrated in Figures 3.4 and 3.7 (pages 45–52).

6.3.2 *Connection Establishment*

Flexible Packaging similarly insulates a ware from differences in the Connection Establishment aspect, since decisions about how an interaction is initiated and terminated are typically encapsulated within the packager. Note, however, that the ware provider determines those component interactions that he wishes to be flexible (for which he uses channels) and those he wishes to fix (for which he makes direct I/O library calls). For example, the PNG ware makes direct I/O library calls to establish a connection to the PNG file, the source of whose name is flexible. A ware is insulated from changes in the Connection Establishment aspect only for those interactions that are flexible and therefore entirely encapsulated in the packager.

6.3.3 *Data Representation*

The use of data conversion routines in the channel maps supports changes in the Data Representation aspect. As previously mentioned, different forms of interaction often involve different data representations; these differences must be overcome when combining a ware with different packagers. Unlike traditional component development, where decisions about data representation can permeate the component's source code, the use of typed channels and channel maps creates a readily identifiable place for the system architect to cope with difference in data representation.

Unfortunately, the use of conversion routines to overcome a difference in Data Representation requires more effort from the system architect than does the use of coroutines to overcome a difference in Data and Control Transfer. The use of coroutines requires the system architect to establish a name mapping between different coroutines' channels; the use of conversion routines requires the system architect to write source code.

Two experiences from the experiments mitigate the apparent effort. First, pervasive infrastructures can establish vocabularies of datatypes to which packagers and wares agree. For instance, with the PNG components, the Microsoft Win32 library provides a set of datatypes for windowing and graphics that the ware and packagers share in common. Having such established vocabularies of datatypes decreases the need for conversion routines. Second, the system architect can build up a library of conversion routines and use the library in the channel map. For example, several of the components require a conversion between the BSTR and char* representations of the type string. Because of this, I created a UniCon property whose value is this conversion routine and referred to that property in several of the channel maps.

The channel maps, as presented in this thesis, do not make use of such abbreviations, in order to make them more self-contained.

6.3.4 *Failure*

Flexible Packaging is less successful in insulating the ware from differences in the Failure aspect. 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.

Without changing the Flexible Packaging tools, there are three strategies for coping with the absence of a notion of failure with *in* and *out* statements. The first is the use of additional channels to model failure. A ware could model a possibly failing *in* statement as an *alt* statement that contains the original *in* statement, plus an *in* statement on an error-reporting channel. Similarly, a ware could model a possibly failing *out* statement as an *out* statement followed by an *in* statement that reports whether the *out* failed. The problem with this approach is that it reduces the compatibility between wares and packagers. For example, a possibly failing *out* channel should be compatible with a reliable (never failing) *in* channel. Using the extra-channel strategy, the possibly failing *out* channel would be modeled with two channels (an out channel, plus an in channel that reports the success or failure of the out channel). The system architect would then face the task of mapping two channels to one.

The second strategy is to supplement channels with other language features, like exceptions. For example, an *in* statement that possibly fails could be modeled as an *in* statement surrounded with *try/catch* brackets to catch an exception representing the failure of the *in* statement. The disadvantage of this approach is that the channel signatures and channel maps would have to be supplemented with documentation about the exceptions, documentation that would not affect the check for channel signature compatibility.

The third strategy is to have a channel's type include a value that represents failure, in the style of the option type in Standard ML or sentinel values like NULL or Nil in other languages. This turns the question of compatibility between possibly failing channels and reliable channels into a datatype compatibility question, for which channel maps provide a solution. The disadvantage to this approach is that the use of failure values can turn the source code into a rat's nest of conditional statements – a problem that exceptions often improve. Finding an appropriate way to incorporate a notion of failure into channels is future work.

6.3.5 *State Persistence and State Scope*

Finally, Flexible Packaging does not insulate the ware from differences in the State Persistence or State Scope aspects. 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 the State Scope aspect, the abstract type would support create and destroy operations; to support 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.

6.4 VALIDATION OF THE THESIS CLAIM

In Chapter 1, the dissertation makes the following claim:

It is possible to separate a software component's functionality from its packaging so that the following properties hold:

1 **The system architect can independently choose a software component's functionality and packaging at the time the component is to be integrated into a system.**

2 **She can readily determine whether her choices of functionality and packaging are compatible.**

3 **If her choices of functionality and packaging are compatible, she can automatically produce a software component based these choices. This**

component's performance is comparable to the equivalent component developed by hand.

4 **Her functionality and packaging choices are compatible often enough to allow her to produce a useful variety of components.**

5 **The component provider and system architect expend less development effort on the whole using this approach than they would using traditional development methods.**

The Flexible Packaging method achieves exactly such a separation of concerns. Using evidence from the experiments, the following sections describe how Flexible Packaging provides each of the properties that the claim demands.

6.4.1 *Selecting functionality and packaging during system integration*

The system architect chooses a software component's functionality by selecting a ware. She chooses the component's packaging by acquiring a packaging generator for her desired type of packaging and by writing a packaging description that contains the information that the generator requires. Both selecting the ware and writing the packaging description are done at system integration time. Writing the packaging description at system integration time allows the system architect to tailor the component's interactions to the specific needs of the integration context.

6.4.2 *Determining the compatibility of functionality and packaging*

Once the system architect has written a packaging description, she runs the Packgen tool on it to produce a packager. The ware and the packager each have a channel signature, which allows the compatibility of the ware and the packager to be checked. To check compatibility, the system architect first creates a channel map to establish a correspondence between the two channel signatures. In the process of establishing this correspondence, she may discover that a channel in one of the modules has no corresponding channel in the other module. For example, were she to package the PNG ware as a filter, she would discover that the ware has a channel

```
channel in struct {  struct { long left, bottom, top, right; }* rect;
                     void* hdc; } Paint;
```

that communicates graphics information, for which there is no corresponding channel in the filter packager. The inability to find a correspondence between the ware and packager's channel signatures signals that the functionality and packaging are incompatible.

If she can establish the correspondence, the Ciao compiler uses the correspondence to check the compatibility of the two channel signatures. This check catches more subtle incompatibilities that lead to run-time deadlock. If the ware's and packager's channel signatures pass this check, no run-time deadlock can occur, and the ware and packager – and hence the functionality and packaging – are compatible.

6.4.3 *Producing a component with reasonable performance*

Given a ware and a packager with compatible channel signatures, the Ciao compiler automatically integrates them to form the final component. The only difference between the source code of a component produced through Flexible Packaging and that of an equivalent component produced by hand is the use of channels to integrate the packager and the ware. In a typical traditional component, the source code for the packager and the ware would be intermingled; there would be no explicit interface between these pieces of code, hence no performance penalty for crossing an interface boundary. With Flexible Packaging, there is an explicit interface between the two pieces of code, one based on shared data and coroutines. As the timing information in Section 5.3.2 demonstrates, the performance penalty for this interface is modest.

6.4.4 *Producing a useful variety of components*

As the experiments indicate, I was able to produce a useful variety of components from each ware. There are two sense in which the variety is useful. First, the nine experimental packagings that I used to build the components are drawn from today's practice, and hence useful to today's developers. In this sense, the components are individually useful. Further, I was able to package each ware with several packagings and, in some cases, with packagings that represent competing component markets. For example, I was able to package the PNG component as both a Netscape plug-in and an ActiveX control, two component standards that directly compete. Were I a component vendor, I would be able to sell PNG components both in the plug-in market and in the control market, rather than committing myself to a single packaging-based market, as often happens today. Hence, I was able to produce a variety of components, where the variety would allow me to compete in multiple markets.

6.4.5 *Producing a component with reasonable e ort*

The data in the experiments is too preliminary to use as the basis for quantitative cost estimates. Instead, the experiments can be used to argue that Flexible Packaging addresses both the direct cost and opportunity cost associated with component integration in current practice. As introduced in Chapter 1, when a component integrator must overcome a packaging mismatch, he faces the direct cost of producing glue code or of employing some other fix. In the case where the component's packaging matches the integration context, he faces the opportunity cost of disregarding those components who packaging does not match.

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 system architect'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. Thus, while it is too soon to measure the cost benefit of using Flexible Packaging, the method does address both the direct cost and opportunity cost associated with current component integration.

6.5 TOWARD FLEXIBLE PACKAGING IN PRACTICE

This dissertation's research focusses on the creation of Flexible Packaging and the validation of its feasibility. Although at this early stage of the research it is too soon to describe the practice of using Flexible Packaging, here are a few observations about a path to industry adoption, the design of wares, and the effect of the method on the semantic mismatch problem.

6.5.1 *Adopting Flexible Packaging*

One of Flexible Packaging's disadvantages is that a number of developers must adopt it before it system architects can reap its benefits. In particular, until component providers begin to provide wares instead of fully packaged components, system architects will be unable to find wares in the market that have the functionality that they require. Further, component providers will be uninspired to sell wares until system architects demonstrate a market for them. The way forward from this chickenand-egg problem is the use of Flexible Packaging in-house to produce traditional components.

Currently, many component providers either produce components only to a single packaging standard (say, Corba) or bear the expensive of producing a family of components that differ in their packagings (say, both Corba and ActiveX). For developers in this situation, Flexible Packaging lowers the cost of producing this family. A component provider could use Flexible Packaging in-house to produce wares. Rather than selling these wares directly or in addition to it, the component providers could write their own packaging descriptions and sell the resulting fully packaged components. For those developers that currently sell a family of components, using Flexible Packaging would likely lower their costs: it would provide packaging generators for packagings that currently have no tools (e.g. Netscape plug-ins); it would bring uniformity to the process of packaging components rather than relying on in-house packaging gurus; and it would reduce their version control problems by allowing a component's functionality to be expressed in a single source module rather than one version per packaging. Finally, by using Flexible Packaging in-house, component providers would be prepared to sell wares directly, when and if the Flexible Packaging community reaches critical mass.

6.5.2 *Composing and decomposing wares*

Traditional module systems support a uniform type of compositionality: smaller modules are composed to form large modules; larger modules decompose into smaller modules. This uniformity, however, does not scale up to the largest systems. Although a 10,000-line module may be composed of 10 1000-line modules, typically a 10,000,000-line system is not one large module composed of 10 1,000,000-line modules. At some point while scaling up, modules are not composed to form a larger module, but instead to form a component. Whereas interaction between modules is through mechanisms supported by the programming language in which the modules are written (e.g. function call, method call, shared variables), interaction between components is through such other mechanisms as pipes, messages, and shared databases. While composing parts to form larger parts, at some point the architect decides that the interaction among peer parts should no longer be through programming language constructs, but through other mechanisms. Traditionally when a developer decides that a module will be a component, he commits to the mechanism by which the component interacts with other components.

Flexible Packaging changes this notion of composition. By supplementing the programming language mechanisms by which module interact with a new mechanism – channels – a ware can be composed with another ware in two ways. First, the wares can interact directly through channels, i.e., the wares can be treated as modules, which are composed to form a larger module. Second, each ware can be packaged to form a component, and the components can then be composed to form a larger component or system. The choice between these two types of composition depends on the interaction mechanism that the integrator finds most suitable. For example, if there is a data flow between the two wares (one ware does an *out*, the other does an *in*), then one possibility is to place a channel between the *out* and the *in*. However, if the architect wants the communicated data to be persistent, for example, he could implement the data flow by packaging one ware to be a file writer and the other to be a file reader. The person integrating the wares chooses between these two types of composition by choosing the interaction mechanism that has the properties she wants.

In short, Flexible Packaging improves the uniformity of composition. At the time of developing a traditional developer writes his source code, he chooses how his part will interact with other parts. If he chooses a mechanism that the programming language supports, his part is called a module; if he chooses mechanisms likes pipes, messages, or databases, the part is called a component. With Flexible Packaging, the same ware may either be treated as a module (interaction through channels alone) or as a component (interaction through a packager). Unlike with traditional development, the ware provider need not anticipate which style of composition will be used.

6.5.3 *Semantic mismatch*

The problem of packaging mismatch is very related to the problem of semantic mismatch. The former occurs when components disagree about the mechanism they use to exchange data and control; the latter, when components disagree about concepts that are engendered in the data that they exchange. Since the type or format of the data that components exchange attempts to reflect the meaning of the data, the boundary between the two problems is blurry. Consider the example mentioned in Chapter 1. Two inventory databases are to be integrated. Both represent the number of items on the shelf: one counts individual items (a carton of eggs counts as 12); one counts units of items (a carton of eggs counts as 1). If the databases both use the same type to encode the counts (e.g. type Natural), then the problem is semantic mismatch; if they use different types (e.g. type Natural versus type Unit-Natural), then the problem is packaging mismatch. With packaging mismatch, the problem is manifest in the computing artifacts (for example, in the declared types of the data) and is therefore amenable to automated or semi-automated solutions. With semantic mismatch, the problem is implicit and therefore requires human intervention to solve. This difference in the classes of applicable solutions is the main reason for drawing the distinction between the two problems.

Although addressing semantic mismatch is beyond the goal of the Flexible Packaging research, Flexible Packaging does somewhat ameliorate semantic mismatch by making a component's interactions more explicit. A given data item, as it enters or leaves a component, is mentioned in up to four places: in the packaging description (how the data is formatted for exchange with other components); in the packager's channel signature (how the packager exchanges the item with the ware); in the ware's channel signature (how the ware exchanges the item with the packager); and, if the ware and packager disagree about the data's type, in the channel map. Unlike with a traditional software component, where a data item's type (or, indeed, its presence at all) can be buried within the source code, Flexible Packaging makes explicit both how the component types the data, both with respect to its functionality and its interactions. This explicit treatment at least makes the presence of data exchange manifest. In the end, however, the deep problem of semantic mismatch – the fact that formatting or typing does not always encode semantic intent – is one that Flexible Packaging does not address.

6.6 FUTURE DIRECTIONS

With the basic feasibility of the Flexible Packaging method established, there are several direction in which to take the research.

6.6.1 *Examining the human factors of module integration*

Some of the questions that this dissertation addresses are not questions about computations, but questions about people. For example, consider a question such as this: which modularity mechanism allows a programmer use the least effort to combine independently authored modules? This kind of question cannot be answered by examining modularity mechanisms in isolation, but requires an examination of how people use these mechanisms in practice. Bringing the controlled studies and ethnographic techniques from human–computer interaction research to bear on module composition problems would be an interesting direction in which to take this research.

For instance, the design of channels is intended to allow a developer to compose two or more independently authored modules into a combined computation. The experiments in Chapter 5 demonstrate that a developer can use channels to combine two or more modules. However, the experiments do not demonstrate whether channels support multiple independent authors, since a single developer created all of the experimental modules. To validate the use of channels to support independent authors, I intend to run a controlled experiment.

The experiment involves mixing and matching modules that several programmers independently create. Each programmer would be given a short tutorial on the use of channels and the same well specified programming task. The task would be to create a software component with specific functionality and a specific packaging, by developing and combining a ware and a packager. Because each programmer would develop both the ware and the packager, combining them would not require a channel map or additional coroutines. Indeed, to keep learning to a minimum, the experiment's tutorial would not mention these features. The key to the experiment is that each programmer himself designs the channel interface between the ware and packager. Together, the n programmers create n wares and n packagers. With their programming tasks complete, I, as the experimenter, would try all n^2 combinations of wares and packagers to test whether they can be combined at all and, if so, whether I need channel maps or additional coroutines to combine them. (To reduce the needed manpower in the face of the quadratic number of combinations, the experimenters could examine a random subset of the possible combinations or the subjects themselves could be asked to try combinations. The latter possibility engenders a potential learning confound.) Witnessing the kinds of channel mismatches that occur during the experiment would provide either validation that channels are sufficient to support independently authored modules or would point out where channels need improvement.

Studying developers under controlled conditions is not the only interesting approach, however. In some cases, developers leave behind a sufficiently rich trail in their development effort that an ethnographic approach could be revealing. For example, consider this question: What are the kinds of modifications that developers make in their software for which current modularity mechanisms (e.g. procedural interfaces) keep the modifications isolated? What are the kinds of modifications for which the modularity mechanisms allow the changes to be pervasive? One might guess that changes in data representation that are encapsulated within abstract datatypes and changes to algorithms encapsulated within procedures would be well isolated; whereas, that changes to supporting infrastructure, like memory management, would not.

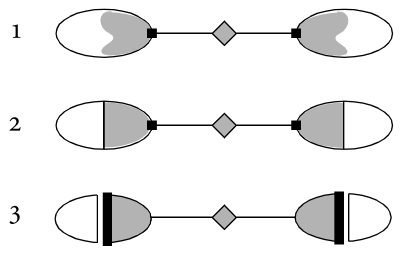
Examining the revision history of actual software projects could provide concrete evidence to support such conjectures, as well as a possible classification of the software evolution problems for the research community to solve. For example, I would examine the revision history documentation and the actual software revisions of a well documented project, like many of those from the Open Source movement. Using the revision history documentation to understand the motivation behind each change to the software, I would judge the degree of isolation for each change and look for correlations between the degree of isolation and both the type of change (change in data representation, change in algorithm, change in substructure) and the motivation for the change (cross-platform port, bug fix, feature addition). Even with good tools, such studies could take considerable effort, given the messiness of real-world development projects. However, having evidence from actual projects provides the material both to answer important questions about the human factors of software composition and to classify the research problems to be solved.

6.6.2 *Supporting the rapid development of system architectures*

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; connectors mediate the interaction among components.[49] As mentioned in Chapter 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 can be attached to a pipe connector, but not a procedure call connector or a socket connector; a procedure library can be attached to a procedure call connector, but not a pipe or socket connector; an internet server component can be attached to a socket connector, but not a pipe or procedure call connector. Creating type systems that capture the connectivity limitations of current practices is one of the main achievements of ADLs.

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. This dissertation advocates a process for the system architect that is the least disruption from current practice. The system architect 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 architect can follow current practice, namely composing the component with connectors to form the final system.

Instead, I intend to explore a new ADL in which the component/connector separation coincides with the functionality/interaction separation. The evolution to this new ADL can be pictured as follows:



In the jargon of ADLs, the black square represents the attachment between a role and a port/player.

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 (grey). Notice that the component/connector distinction (at the black squares) does not coincide with the functionality/interaction distinction (at the grey/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, 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; and so on. 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 a data structure, an architect could insert a database connector between them, which provides non-interference 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.

6.6.3 *Decomposing functionality*

The research in this dissertation concentrates on separating a component's functionality from its packaging. However, a component's functionality itself should be decomposed into smaller pieces – not as is done in practice today, with the focus on implementation benefits (e.g. using the principle of information hiding), but instead in a way to benefit the work practice of ordinary computer users.

Two trends in modern desktop computing demonstrate the degree to which we are currently failing to achieve useful software decompositions. First, common desktop applications, like document editors and spreadsheets, have become bloated with features that many users either never use or use only for occasional tasks. The monolithic nature of these applications makes them difficult to learn and has caused a proliferation of related, but separate tools for similar tasks. Consider document editing on a Windows desktop machine. The Microsoft Windows tools Notepad, Wordpad, and Word are all intended to support document editing as a primary activity and are designed for users or tasks that vary in sophistication. Further, other tools like email programs, web forms, and development environments are primarily designed for a different task but nonetheless involve document editing. All of these document editors involve a somewhat different look-and-feel and different editing capabilities.

The second trend is that people are increasingly incorporating both applications and data from the web into their daily computing tasks. Due to the web's frontier nature, web users confront myriad interface styles (email-based services, applets, chat rooms, ActiveX components, web forms, and bots, to name a few), a trend reminiscent of the chaos before standardized desktop platforms like Macintosh and Windows. In short, users today face both many application styles and applications that are internally complex.

To address this, I would decompose the monolithic notion of an application into a uniform, extendable vocabulary of capabilities that can be combined to suit a particular user's knowledge and current task. For example, someone learning about spreadsheets would start with a simple grid of numbers and formulas and later add capabilities like macros and graphs when and if he chooses to learn about them. As another example, a user creating a document might start only with text editing capabilities and later add communication capabilities, if he decides to share the document with colleagues, or typesetting capabilities, if he decides to publish the document. The uniform nature of the vocabulary would mean the same capability would support multiple work contexts. For example, the capability of editing a table of numbers would be the same whether in the context of a document or a spreadsheet.

This new desktop scenario means rethinking traditional monolithic user interface abstractions, like windows and pull-down menus, which are geared toward completely exposing a fixed set of capabilities. It further means rethinking traditional software decomposition methods, which make some decisions hard to change because they either cut across module boundaries or globally affect a module. For example, a module (such as an ActiveX control) for document editing would support very different internal representations depending on whether it supports such features as the following: spell checking, with the need to represent words, not just characters; line wrapping, with the need to represent lines, not just words; typographic styles, with the need to represent the appearance of words; and justification, with the need to represent the size of characters and the context in which the text appears. Each of these variations on a document editing module would also support different operations to allow access to the different capabilities. We need a new approach to modularity that would allow an individual feature, like typographic styles, to be encapsulated so that it could be added or removed without effecting the other features. In the easier case, a feature could be added at integration time to allow a developer to create a family of products that vary by feature set. Conquering the static case would then allow us to build applications with feature sets that vary at run time, as described above.

6.7 SUMMARY

This dissertation presents a new approach to developing and integrating software components that represents a better distribution of responsibilities than that of current practice. Currently, a component provider is responsible both for the component's functionality – the source of its economic value – and the packaging that makes it available for integration with other components. Achieving this packaging often requires the component provider to understand a number of details about the packaging, like the API to an interaction library or the use of packaging-specific construction tools. Becoming proficient with sophisticated component packagings, like COM or Corba, often requires months of training, a level of investment that discourages component providers from learning about many packagings. In contrast, a ware provider need only learn about channels, a mechanism far simpler than packagings like COM or Corba.

To relieve the ware provider from learning about packaging details, the Flexible Packaging method introduces the role of the packaging specialist. There is one packaging specialist for each type of interaction between components – one for COM, one for Corba, one for Netscape plug-ins; hence no more than a few dozen people fill this role. A packaging specialist designs two interfaces. First, he reviews the steps needed to give a component his kind of packaging and selects those decisions about the packaging that are fundamental. These fundamental decisions he defers to the system architect. Second, he designs the interface to the packager, reducing the details of his style of interaction to a set of channel operations. Designing both of these interfaces requires knowledge and thoughtfulness, but only a few people need fulfil this high-effort responsibility.

Finally, Flexible Packaging also changes the nature of the system architect's responsibilities. Today, a system architect must often be very familiar with different component packagings, both to detect whether his integration task involves packaging mismatch and to resolve packaging mismatches when they occur. For example, overcoming a packaging mismatch between two components by building a bridge to place between them typically requires detailed knowledge of the two types of packagings being bridged. In contrast, Flexible Packaging requires no detailed knowledge about component packaging. To give a ware a particular packaging, the system architect makes the high-level decisions about the packaging; the packaging specialist takes care of the rest. To combine the packager and the ware to form the final component, the architect needs to understand channels and how to overcome their differences through channel maps and additional coroutines, knowledge which is gained once and used in every integration task.

This dissertation presents a new approach to developing and deploying software components that greatly reduces one of today's biggest integration problems, packaging mismatch. Currently, a system architect faces either limitations or hard work when assembling a system from parts: she can limit her choice of architectures to those that fit the parts that she is reusing; she can limit her choice of parts to those that fit the architecture that she is using; or she can create "glue code" to overcome mismatches between the interaction styles of the components and the architecture. Instead, in a world in which components are flexibly packaged (i.e. where all components are wares), the system architect chooses a software architecture and writes packaging descriptions to tailor the reused components to the architecture. In the case where a component cannot be tailored because the generated packager is incompatible with the ware, the incompatibility is fundamental – for example, because the packaging promises a value that the computation cannot not deliver. Hence, although Flexible Packaging cannot be said to eliminate the problem of packaging mismatch completely, it reduces the problem down to fundamental mismatches between the architect's architectural and functionality choices. In the world of flexibly packaged components, the architect concerns herself with whether the architecture and components fit the problem at hand, and not with the details of the packaging technology of the day.

Although Flexible Packaging was designed to address packaging mismatch, it may also prove helpful to addressing other integration problems. For example, as previously mentioned, making all of a component's data requirements explicit in its channel signature may help expose semantic mismatch problems. Further, since channel signatures provide a uniform interface to all components, they potentially provide a basis for searching for components in repositories. Other problems are not particularly different in a world of flexibly packaged components. For example, the functionality that a flexibly packaged component provides may differ from the needed functionality (a near-miss) or may be inconsistent with the component's documentation. Flexible Packaging makes these problems no better and no worse. In short, while Flexible Packaging does not address some integration problems, it does provide a setting for addressing others and nearly eliminates one of them, the problem of packaging mismatch.

References

1 Robert Allen.
A Formal Approach to Software Architecture.
Dissertation, Carnegie Mellon University, 1997.
Cited on page 59.

2 Don Batory, Bernie Lofaso, and Yannis Smaragdakis.
JTS: Tools for Implementing Domain-Specific Languages.
In *Proceedings of the Fifth International Conference on Software Reuse, Victoria, Canada*, 1998, pages 143–153.
Cited on page 87.

3 A.D. Birrell and B.J. Nelson.
Implementing remote procedure calls.
Cited on page 88.

4 Andrew P. Black.
An asymmetric stream communication system.
In *Proceedings of the Symposium on Operating Systems Principles*, 1983, pages 4–10.
Cited on page 46.

5 Barry W. Boehm and William L. Scherlis.
Megaprogramming.
In *Proceedings of the Software Techonology Conference*, DARPA, 1992.
Cited on page 1.

6 Kraig Brockschmidt.
Inside OLE.
Microsoft Press, second edition, 1995.
Cited on page 29.

7 John Callahan.
Software Packaging.
Dissertation, University of Maryland, 1993.
Cited on page 57.

8 John R. Cameron.
An overview of JSD.
Transactions on Software Engineering 12(2):222–240.
Cited on page 59.

9 Nicholas Carriero and David Gelernter.
Linda in context.
Communications of the ACM 32(4): 444–458, April 1989.
Cited on page 57.

10 David Chappell.
Understanding ActiveX and OLE: A guide for developers and managers.
Microsoft Press, 1996.
Cited on page 36.

11 Melvin E. Conway.
Design of a separable transition-diagram compiler.
Communications of the ACM 6(7): 396–408, 1963.
Cited on pages 48 and 58.

12 Robert DeLine.
Avoiding packaging mismatch with Flexible Packaging.
In *Proceedings of the International Conference on Software Engineering*, *Los Angeles*, *California*, 1999, pages 97–106.
Cited on page 13.

13 Robert DeLine.
A Catalog of Techniques for Resolving Packaging Mismatch.
In *Proceedings of the Syposium on Software Reusability*, *Los Angeles*, *California*, 1999, pages 44–53.
Cited on page 21.

14 Robert DeLine.
Toward user-defined element types and architectural styles.
In *Proceedings of the Second International Software Architecture Workshop*, *San Francisco*, *California*, 1996, pages 47–49.
Cited on page 72.

15 Robert DeLine, Gregory Zelesnik, and Mary Shaw.
Lessons on converting batch systems to support interaction.
In *Proceedings of the International Conference on Software Engineering*, *Boston*, *Massachussets*, 1997, pages 195–204.
Cited on page 43.

16 Chrysanthos Dellarocas.
Toward a design handbook for integrating software components.
In *Proceedings of the Symposium on Assessment of Software Tools and Technologies*, 1997, pages 3–13.
Cited on page 34.

17 Edsger W. Dijkstra.
The structure of the "THE" multiprogramming system.
Communications of the ACM 11(5): 341–346, May 1968.
Cited on page 44.

18 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design patterns: Elements of reusable object-oriented software.
Addison-Wesley, 1994.
Cited on page 17.

19 David Garlan, Robert Allen, and John Ockerbloom.
Architectural mismatch, or why it's hard to build systems out of existing parts.
In *Proceedings of the Seventeenth International Conference on Software Engineering, Seattle, Washington*, April 1995, pages 179–85.
Cited on pages 1, 33, and 155.

20 David Gelernter and Nicholas Carriero.
Coordination languages and their significance.
Communications of the ACM 35(2): 97–107, February 1992.
Cited on page 57.

21 Richard Grimes, Alex Stockton, George Reilly, and Julian Templeman.
Beginning ATL COM Programming.
Wrox Press, Birmingham, England, 1998.
Cited on page 36.

22 J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch family of specification languages.
IEEE Software 2(5): 24–36.
Cited on page 44.

23 C.A.R. Hoare.
An axiomatic basis for computer programming.
Communications of the ACM 12(10):576–580 and 583, October 1969.
Cited on page 44.

24 C.A.R. Hoare.
Proof of correctness of data representation.
Acta Informatica 1(4):271–281, 1972.
Cited on page 44.

25 C.A.R. Hoare.
Communicating Sequential Processes.
Prentice-Hall International, London, 1985.
Cited on pages 10, 63, and 65.

26 Inmos Ltd.
occam Programming Manual.
Prentice-Hall International, 1984.
Cited on page 49.

27 M.A. Jackson.
Principles of Program Design.
Cited on page 153.

28 Bill Janssen, Mike Spreitzer, Dan Larner, and Chris Jacobi.
The ILU 2.0 Reference Manual.
ftp://ftp.parc.xerox.com/pub/ilu/2.0a13/manual-html/manual_toc.html
Cited on page 28.

29 Gilles Kahn and David B. MacQueen.
Coroutines and networks of parallel processes.
In *Proceedings of the IFIP Congress*, pages 993–998, 1977.
Cited on page 59.

30 Rick Kazman, Paul Clements, Len Bass, and Gregory Abowd.
Classifying architectural elements as foundation for mechanism mismatch.
In *Proceedings of the International Computer Software and Applications Conference*, 1997, pages 14–17.
Cited on page 33.

31 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin.
Aspect-oriented programming.
In *Proceedings of the European Conference on Object-Oriented Programming, Finland*, Springer-Verlag LNCS 1241. 1997, pages 220–242.
Cited on page 58.

32 Cristina Videira Lopes and Gregor Kiczales.
D: A language framework for distributed programming.
Technical report SPL97-010, P9710047 Xerox Palo Alto Research Center, February 1997.
Cited on page 58.

33 M. D. McIlroy.
Mass produced software components.
In *Proceedings of the NATO Software Engineering Workshop, Garmisch, Germany*, 1968, pages 138 – 150.
Cited on page 1.

34 Robin Milner, Mads Tofte, Robert Harper, David MacQueen.
The definition of Standard ML.
MIT Press, revised edition, 1997.
Cited on page 73.

35 Diane E. Mularz.
Pattern-based integration architectures.
Chapter 7 in James O. Coplien and Douglas C. Schmidt, editors, *Pattern languages of program design*, 1995. Addison-Wesley.
Cited on page 34.

36 Bertrand Meyer.
Object-oriented Software Construction.
Prentice-Hall, 1997.
Cited on page 43.

37 John Ockerbloom.
Mediating among diverse data formats.
Dissertation, Carnegie Mellon University, 1998.
Cited on page 27.

38 Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, Jeffrey Ullman.
A query translation scheme for rapid implementation of wrappers.
In *Proceedings of the International Conference on Deductive and Object-oriented databases*, 1995, pages 161–86.
Cited on page 26.

39 D.L. Parnas.
On the criteria to be used in decomposing systems into modules.
Communications of the ACM 15(12): 1053 – 8, December 1972.
Cited on page 44.

40 D.L. Parnas, P.C. Clements, and D.M. Weiss.
The modular structure of complex systems.
IEEE Transactions on Software Engineering SE-11(3): 259 – 66, March 1985.
Cited on pages 3 and 44.

41 James M. Purtilo and Joanne M. Atlee.
Module reuse by interface adaptation.
Software–Practice and Experience 21(6): 539–56, 1991.
Cited on page 24.

42 John H. Reppy.
Concurrent programming in ML.
Cambridge University Press, 1999.
Cited on pages 59 and 96.

43 John H. Reppy.
CML: A higher-order concurrent language.
SIGPLAN Notices 26(6):293–305, June 1991.
Cited on page 59.

44 A.W. Roscoe.
Model-checking CSP.
In *A Clasical Mind*, *Essays in Honour of C.A.R. Hoare*, Prentice-Hall, 1994.
Cited on page 89.

45 Mary Shaw.
Architectural issues in software reuse: It's not just the functionality, it's the packaging.
In *Symposium on Software Reusability*, 1995, pages 3–6.
Cited on page 33.

46 Mary Shaw and Paul Clements.
A field guide to boxology: Preliminary classification of architectural styles for software systems.
In *Proceedings of the International Computer Software and Applications Conference*, 1997, pages 6–13.
Cited on page 33.

47 Mary Shaw, Robert DeLine, Daniel Klein, Theodore Ross, David Young, and Gregory Zelesnik.
Abstractions for software architecture and tools to support them.
Transactions on Software Engineering 21(4):314–335, April 1995.
Cited on page 71.

48 Mary Shaw, Robert DeLine, Gregory Zelesnik.
Abstractions and implementations for architectural connections.
In *Proceedings of the Third International Conference on Configurable Distributed Systems*, *Annapolis*, *Maryland*, 1996, pages 2–10.
Cited on pages 22, 33, and 78.

49 Mary Shaw and David Garlan.
Software architecture: Perspectives on an emerging discipline.
Prentice-Hall, 1996.
Cited on page 169.

50 Kevin J. Sullivan and John C. Knight.
Experience assessing an architectural approach to large-scale systematic reuse.
In *Proceedings of the Eighteenth International Conference on Software Engineering, Berlin, Germany*, 1996, pages 220 – 229.
Cited on page 1.

51 Katia Sycara, Keith Decker, Anadeep Pannu, Mike Williamson, and Dajun Zeng.
Distributed intelligent agents.
IEEE *Expert* 11(6):36–46, 1996.
Cited on page 27.

52 Tom Wu.
Behind the scenes of the Adventure Web.
http://www-tjw.stanford.edu/adventure/impl.html
Cited on page 20.

53 Daniel M. Yellin and Robert E. Strom.
Interfaces, protocols, and the semi-automatic construction of software adaptors.
In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, *Portland, Oregon*, 1994, pages 176–90.
Cited on page 24.

54 Amy Zaremski and Jeanette Wing.
Signature matching: A tool for using software libraries.
Transactions on Software Engineering and Methodology 4(2):146–70.
Cited on page 1.