To appear in Symposium on Software Reusability, Los Angeles, CA, May 1999

A Catalog of Techniques for Resolving Packaging Mismatch

Robert DeLine
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA 15213-3891
1-412-268-2582
rdeline@cs.cmu.edu

1 ABSTRACT

A problem that often hampers the smooth integration of reused components into a new system is the packaging mismatch problem: when one or more of a component's commitments about interaction with other components are not supported in the context of integration. Although system integrators have faced and surmounted this problem for decades, their experience largely exists as unrecorded folklore and as specific papers in separate research communities — a situation which makes it difficult to systematically understand and solve an instance of this problem. In order to allow system integrators to attack packaging mismatches systematically, what is known about the problem and its solutions must assembled and organized. To take a step in this direction, this paper first discriminates the chief characteristics of component packaging, which are the sources of mismatch. It provides a catalog of techniques to resolve packaging mismatch, organized according to the architectural commitments involved: on-line and off-line bridges, wrappers, intermediate representations, mediators, unilateral and bilateral negotiation, and component extension. Finally, it describes the issues involved in resolving packaging mismatch, aspect by aspect.

1.1 Keywords

component-based engineering, COT and component brokers, interface issues, software architecture and systemic reuse

2 INTRODUCTION

Software developers have long dreamed of building systems by assembling off-the-shelf components. Today's off-the-shelf components not only encapsulate useful functionality but also many commitments about how the component is to interact with other components in the system. These commitments about interaction constitute the component's *packaging*. Integrating a component into a system is significantly easier when the component's packaging matches the style of interaction used in the system.

One strategy for decreasing the need to consider packaging when reusing a component is to build all components to a packaging standard. Having such a packaging standard is often what is meant by the phrase "plug-and-play compatibility." Unfortunately, as the old quip goes, the best thing about standards is that there are so many to choose from. There are many packaging standards in vogue today, including Corba, COM/ActiveX, JavaBeans, Java class libraries, and Microsoft Visual Basic controls. Building a component to one of these standards makes it difficult to integrate the component into a system built to another of these standards. In recognition of this difficulty, products are appearing whose only function is to bridge between components written to different standards, like the Corba/COM bridge and the ActiveX/JavaBean bridge. Hence, so far, standards have not lessened the need to consider a component's packaging when attempting to reuse it.

Furthermore, a lesson from the emerging software architecture community is that no single packaging standard will ever be appropriate for all components [14]. Good system design involves choosing an architectural style that is appropriate to the system. For example, a system that transforms data in a series of identifiable steps might suitably be built in the pipe–filter style. A development environment that is meant to accommodate an open-ended set of analysis tools might suitably be built in the implicit invocation style. Were these analysis tools intended instead to access common data without mutual interference, then a database-centric style would be more appropriate. Each architectural style implies the mechanisms by which components interact, which in turn implies the packaging the components must have. In a pipe–filter system, the components are packaged as filters; in an implicit invocation system, as anonymous event handlers; and in a database-centric system, as database accessors. Because different architectural styles are suited to different problems, there will always be a variety of packagings in the world. Hence a component's packaging, as well as its functionality, will continue to influence whether it can be reused in a new context.

When a system integrator has a component whose functionality is needed but whose packaging is inappropriate, the problem is an instance of *packaging mismatch*. System integrators have faced and surmounted this problem for decades. A typical solution is to interpose "glue code" (sometimes called a wrapper, bridge, mediator, or adaptor) between the component and the system to compensate for the differences

in packaging. Another typical solution, often used when the former solution is inapplicable, is to modify the component's source code to update its packaging to one appropriate to the current system. Unfortunately, the experience of these system integrators currently exists only as unrecorded folklore and as specific technical papers scattered throughout the computer science literature. As witnessed by the number of different words for "glue code," we lack even a consistent, precise vocabulary. In order to allow system integrators to attack packaging mismatches systematically, what is known about the problem and its solutions must assembled and organized, in much the same vein as the patterns community is doing for object-oriented design [5]. This paper takes a step in that direction. In Section 3, it provides a vocabulary for different kinds of packaging mismatch problems by decomposing a software component's packaging into a set of identifiable aspects. Section 4 then presents a catalog of packaging mismatch resolution techniques, with examples drawn from a variety of architectural styles and classified according to the vocabulary from Section 3. Section 5 describes the issues involved in resolving packaging mismatch, aspect by aspect. Section 6 discusses related cataloging efforts.

3 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 will then be used both to categorize the examples in the catalog of techniques in Section 4 and to discuss issues in resolving mismatch on an aspect-by-aspect basis in Section 5.

3.1 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 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.

3.2 Data and control transfer

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. Table 1 shows several common mechanisms used to transfer data and/or control between software components. Because we are interested in the degree to which these mechanisms differ, the table decomposes the mechanism into a set of properties. Each of the mechanisms listed in the first column of the table can be used to transfer data and/or control both into and out of a component; the second column shows whether data and/or control is transferred. For example, the first row indicates that when a component gets an environment variable, data is transferred into the component; when it sets an environment variable, data is transferred out. When two components use one of these mechanisms to transfer data or control, the table's third column indicates whether the transmission is due to the receiver actively requesting the data/control (pull) or due to the sender actively sending the data/control (push).

This table can be used to judge the degree to which mechanisms differ. 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).

3.3 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.

3.4 State persistence

A component may vary in the degree to which it retains state

Interaction mechanism [transfer in/transfer out]	What is transferred	Transmission requested by receiver (pull) or sender (push)
environment variable [get/set]	data	pull
data stream [read/write]	data	pull
socket [read/write]	data	push or pull (depending on usage)
file [read/write]	data	pull
shared variable [get/set]	data	pull
shared memory [fetch/store]	data	pull
database [retrieve/update]	data (control[a])	pull (push[a])
(remote) procedure [call/return]	control + data	push
coroutine [call/return]	control (data[b])	push
exception [catch/throw]	control (data[c])	push
interrupt [receive/send]	control (data[d])	push
event [notification/announcement]	data	push
message [receive/send]	data	push or pull (depending on usage)
(component-owned) property [set/get]	control + data	push

Table 1: Data and control transfer properties of various popular interaction mechanisms

a. Some databases notify accessors of updates with update triggers.
b. Some coroutine systems allow for arguments and results, as with procedure calls.
c. Some languages allow data to be associated with an exception; others do not.
d. For Unix-style interrupts, the signal number is passed to the interrupt handler.

between interactions. For example, procedure-based modules may be either stateful, like an I/O library where internal buffers are retained between procedure calls, or stateless, like a library of trigonometric functions. Objects in an object-oriented language and its larger cousins, Corba components, ActiveX controls, and JavaBeans, all retain their internal state between method invocations. Servers, whether interacting through sockets or RPC calls, may be stateless, like the Sun NFS file server or a web (HTTP) server, or stateful, like the Andrew file server or an FTP server.

3.5 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.

3.6 Failure

Component 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.

3.7 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.

4 CATALOG OF MISMATCH RESOLUTION TECHNIQUES

Following the lead of the patterns community, this section provides a catalogue 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 in a consistent format 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.

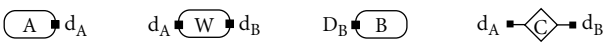
What principally distinguishes one technique from

another are the set of packaging commitments that are made, when they are made, and what architectural element embodies 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.

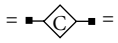
The template uses a schematic diagram to show at a glance how the system architecture is transformed to resolve the packaging mismatch and allow ready comparison of the techniques. Each diagram is a series of rows representing significant times during the development of the system, when either commitments are made or the architecture changes. Borrowing from architectural description languages (ADLs) [14], 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:

Although 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.

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 may 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. 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 commitment-invariant connectors are depicted with equal signs at the roles:

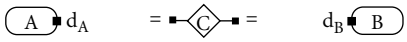


For example, a procedure call connector is indifferent about the number, order, and types of arguments passed from the procedure caller to the procedure definer, so long as the commitment is the same for the caller and definer.

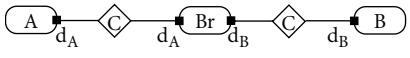
The examples that appear in the templates were chosen, not because they are the best or most representative examples of the technique, but because sufficiently detailed documentation about them is available. Because much of this documentation consists of research papers, there is a bias toward automated 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.

4.1 On-line Bridge

4.1.1 *Schematic*

(s1)


(s2)

(s3)


4.1.2 *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.

4.1.3 *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).

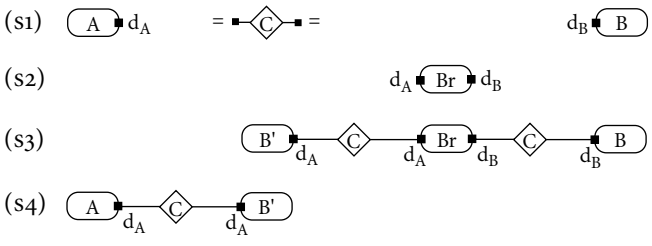
Interpose this component between *A* and *B* (s3). The component *Br*'s computation makes up for the differences between the commitments d_A and d_B. How it does this depends on the aspect of interaction that d_A and d_B represent, which is the topic of Section 5. Although the examples below involve 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.

4.1.4 *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.

4.1.5 *Examples*
- Nimble [10]
  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 [17]
  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

4.2 Off-line Bridge

4.2.1 *Schematic*



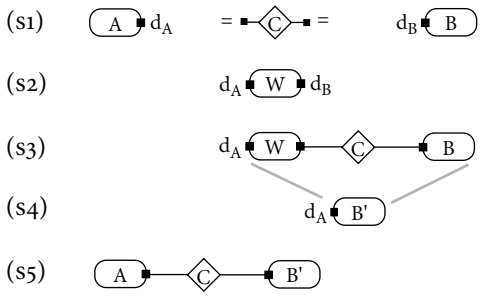
4.2.2 *Problem*
Same as for the On-line Bridge technique, with the restriction that component *B* is some form of persistent data (s1). The mismatch between *A* and *B* is about data representation.

4.2.3 *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 runtime, use the Mediator technique.

4.2.4 *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

4.3 Wrapper

4.3.1 *Schematic*



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

4.3.3 *Solution*

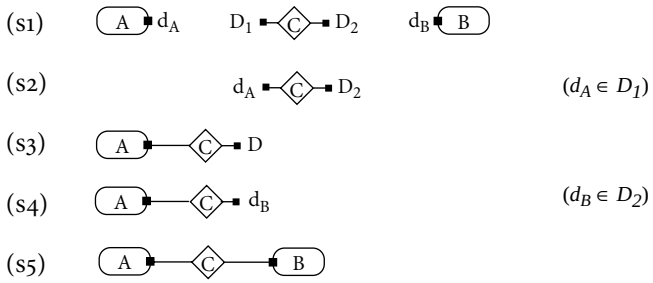
The solution is the same as with the On-line Bridge technique, with one additional step. Before the final integration, encapsulate the wrapper *W* (the analogue of bridge *Br*), the component *B*, and the connector between them within a new component *B*' (s4). This encapsulation step is about both abstraction 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 paper'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.)

4.3.4 *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 address 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 [10]
  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

4.4 Mediator

4.4.1 *Schematic*



4.4.2 *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.

4.4.3 *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.

4.4.4 *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.

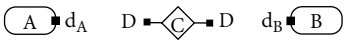
4.4.5 *Examples*

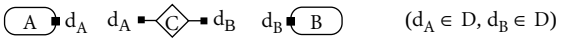
- TOM [9]
  Aspect of packaging: data representation, namely document format
  Component A: a program that reads PostScript documents
  Component B: a document in LaTeX format
  Connector C: the TOM service, which can convert among a variety of document formats by choosing the appropriate conversion tools

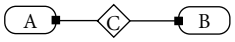
- RETSINA [15]
  Aspect of packaging: data representation, namely the formats of different information sources in the same domain, like stock information
  Component A: a stock portfolio management program
  Component B: a web page showing periodic stock updates
  Connector C: the WARREN system (an instance of the RETSINA framework)

4.5 Intermediate Representation

4.5.1 *Schematic*

(s1) 

(s2) 

(s3) 

4.5.2 *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.

4.5.3 *Solution*

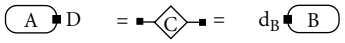
Specialize the connector to the mismatch at hand (s2) and integrate it (s3). Because the set of alternatives is typically fixed when the connector is developed, the system integrator often uses connector-specific tools at system build time to achieve the specialization and integration.

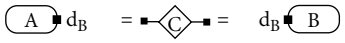
4.5.4 *Examples*

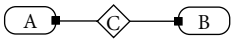
- Xerox PARC's Inter-Language Unification (ILU) [6]
  Aspect of packaging: data representation, namely representation of basic datatypes (integers, strings, booleans, records, etc.)
  Component A: a program written in C
  Component B: a program written in Lisp
  Connector C: the ILU infrastructure, which 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

4.6 Unilateral Negotiation

4.6.1 *Schematic*

(s1) 

(s2) 

(s3) 

4.6.2 *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.

4.6.3 *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.

4.6.4 *Examples*

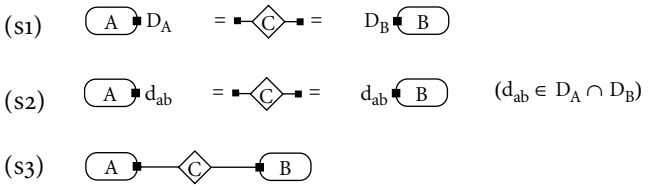
- Microsoft's COM connector
  Aspect of packaging: transfer protocol, namely the interface (collection of procedures) by which A will export computation to B
  Component A: a COM component exporting multiple interfaces
  Component B: a COM component importing a particular interface
  Connector C: the 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

4.7 Bilateral Negotiation

4.7.1 *Schematic*



4.7.2 *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 predetermined channel to choose the alternative) or asymmetric (one component alone chooses the alternative).

4.7.3 *Solution*

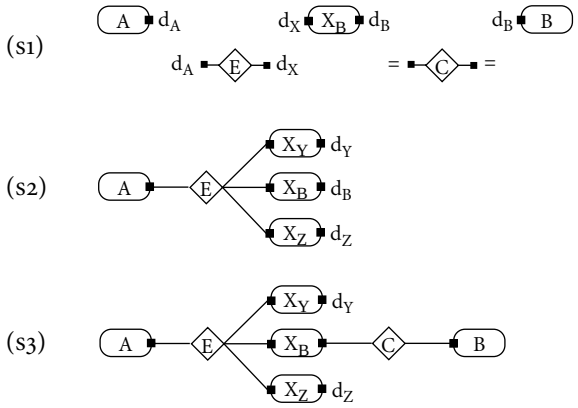
Develop components that support negotiation to prevent packaging mismatch. There are currently too few examples of bilateral negotiation to provide general advice.

4.7.4 *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
  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.
  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)

4.8 Component extension technique

4.8.1 *Schematic*



4.8.2 *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). When *A* initializes itself at runtime, it integrates the extensions (s2). After the extensions are integrated, 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 dynamically integrated, component *A* selects the extension whose commitment agrees with d_B and integrates the extension and *B* (s3).

4.8.3 *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. The technique can be seen as providing 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.

4.8.4 *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)
- Flexible Packaging [2]
  Aspect of packaging: any. Flexible Packaging is a new technique for developing software components that separates a component's functionality from its style of interaction.
  Component A: a module, called a ware, with minimal commitments about interaction
  Extensions X: modules, called packagers, that encapsulate a given style of interaction
  Component B: any
  Connector C: any connector for which there exists a packager supporting the connector

5 RESOLVING PACKAGING MISMATCH, ASPECT BY ASPECT

All of the techniques discussed in the previous section, except Off-line Bridge and Intermediate Representation, are quite general and can be theoretically used to overcome a mismatch in any aspect of packaging. For the On-line Bridge, Wrapper, Mediator, and Component Extension techniques, the system integrator is interposing either a component or connector between the mismatched components, and this interposed element could perform any decidable computation. This section describes the issues that arise when creating this interposed element, aspect by aspect. For ease of discussion, this interposed element will be referred to as a bridge, although the issues apply to the other techniques as well.

For the Unilateral and Bilateral Negotiation techniques, either one or both of the components comes in multiple versions, where the different versions may share a lot of implementation. Understanding how each of the aspects of packaging affects the implementation of this family of related versions is future work.

5.1 Data representation

Bridges that overcome data representation mismatches typically implement an information-preserving transform from one data encoding to another, for example, converting text between EBCDIC and ASCII, integers between big-endian and little-endian representation, and images between GIF and PNG. Other differences in representation have to do with the amount of data. A bridge can turn more data into fewer data, for example, by culling, aggregating, or using a reduction operation like averaging. It can turn few data into more data by using default values or by deriving the missing data from the provided data. As mentioned earlier, some bridges may perform a partial or flawed conversion, such as converting from one word processor document representation to another. Finally, a bridge may not be able to overcome all representation differences. For example, functions and data structures that rely on pointers are notably difficult to represent as byte streams or text, a limitation that arises, for example, in remote procedure call systems.

5.2 Data and control transfer

In order for a bridge to overcome a transfer mismatch between two components, those components must in general agree on the direction of the data/control flow. For example, a bridge can transfer data from a component sending data to one receiving it, but cannot transfer data between two components receiving data. Even when the bridged components agree on the direction of flow, the issue of push versus pull affects the bridge's function: a bridge between an active sender and an active receiver must act as a buffer [1]; a bridge between a passive sender and a passive receiver must act as a pump [1]; and a bridge between an active sender (receiver) and a passive receiver (sender) must pass data/control through and idle otherwise.

5.3 Transfer protocol

In general, the bridge must respect the number and order of interactions that each of the components supports. The more constraints the components place on the interactions, the less freedom the bridge has for supporting different number or order of interactions. Whether a component places many constrains on the number and order of interactions may depend not only on its packaging but on its functionality. For example, a message-passing component is often implemented to accept a fixed message protocol; the only flexibility it supports is that inherent in the protocol. On the other hand, the order in which the sort filter produces output depends on its functionality, not its packaging.

5.4 State persistence

When a component exhibits a state persistence mismatch

because its state persists longer than is needed, the bridge overcoming this problem is left with a garbage collection problem. Consider Wu's effort to make the terminal-based interactive fiction game Adventure available for play over the web as a CGI script [16]. The original game interacts with a single player, prompting for moves and preserving the game state until the player quits. The lifetime of a CGI script is only that of a single URL fetch, much shorter than the indefinite lifetime of the terminal-based game. A bridge between the CGI script and the terminal-based game would have to keep a collection of terminal-based games running, one per web user. Since the web server that runs the CGI script cannot keep an unbounded number of terminal-based games running at once, the CGI script would need a means to garbage collect "unneeded" games. On the other hand, if a component's state persists for too short a time, a bridge may be able to mask the problem if the component provides access to its state (so that it can be saved and restored) or if a component's interaction is batch (stateless) in nature.

5.5 State scope

In general, it is difficult for a bridge to overcome differences in state scope since the effects of interactions are encapsulated within the components it connects. If the scope of an interaction's effect on a component needs to be wider (affecting more internal state), the bridge may be able to repeatedly use an interaction with a smaller scope. For example, if a document editing component provides no print-document operation but does provide a print-page operation, a bridge could mimic the former operation by repeatedly using the latter. Narrowing the scope of an interaction's effect on a component is harder still. One possibility is to replicate the component. If a document editing component provides a print-document operation but no print-page operation, the bridge could replicate the document as several smaller documents, with one page per document, and apply the print-document operation. State scope mismatches are often overcome by redeveloping the component in question [1].

5.6 Failure

A bridge can trivially make a non-failing interaction appear as a possibly failing interaction (the possibility is simply never realized). However, a bridge cannot make a possibly failing interaction appear as a non-failing interaction, unless the failure can be ignored or recovery added. The most obvious case in which a failure cannot be ignored is when the interaction is meant to read data; if the bridge cannot recover from the failure, then it has no way to produce the data that was intended to be read. For example, reading data over a network cannot readily be repackaged as reading a local variable, since the latter cannot fail while the former can fail due to network communication errors.

5.7 Connection establishment

There is nothing a bridge can do if a component exhibits a connection establishment mismatch because it is overcommitted with respect to the set up of a mechanism. For example, if a component reads a file whose name is hard-coded in the component's source code, there is nothing a bridge can do if the component must read some other file in order to be integrated. In the case that the components to be reused have made commitments about architectural connections at the time of their development, it may not be possible to integrate the bridge itself into the system.

6 RELATED WORK

This paper builds on the work of the software architecture community, which has argued for making the types of interaction among components first-class abstractions [14]. In particular, Shaw argued that extra-functional properties of software components, like packaging, often play as important a role during system integration as functional properties [12]. She provided a preliminary list of packaging mismatch resolution techniques and called for it to be "elaborated and refined." This paper is in response to that call.

In a similar vein, several 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 [13]. Mularz reports several patterns (problems paired with solutions) that cover the use of various types of glue to solve integration problems [8]. Dellarocas created an initial handbook of system integration problems paired with solutions, more comprehensive in scope than Mularz's. Dellarocas' classification of interaction problems is broader than this paper's (including, for example, timing dependencies between components) but less detailed in the regions of overlap. Kazman, Clements, Bass, and Abowd classified software components and the interactions among them (connectors) by both how they compose to form systems and how they behave at runtime. Like this paper, their model of runtime behavior is based on the transfer of data and control among components, though this paper views this behavior in more detail.

7 REFERENCES

[1] Andrew P. Black. "An asymmetric stream communication system." In *Proc. Symposium on Operating Systems Principles*, 1983.

[2] Robert DeLine. "Avoiding packaging mismatch with Flexible Packaging." To appear in *Proc. International Conference on Software Engineering*, 1999.

[3] Robert DeLine, Gregory Zelesnik, and Mary Shaw. "Lessons on converting batch systems to support interaction." In *Proc. International Conference on Software Engineering*, 1997.

[4] Chrysanthos Dellarocas. "Toward a design handbook for integrating software components." In *Proc. Symposium*

on Assessment of Software Tools and Technologies, 1997.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*, 1994. Addison-Wesley.

[6] 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

[7] Rick Kazman, Paul Clements, Len Bass, and Gregory Abowd. "Classifying architectural elements as foundation for mechanism mismatch." In *Proc. International Computer Software and Applications Conference*, 1997.

[8] 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.

[9] John Ockerbloom. *Mediating among diverse data formats*. Dissertation, Carnegie Mellon University, 1998.

[10] Yannis Papakonstantinou, Ashish Gupta, Hector Garcia-Molina, Jeffrey Ullman. "A query translation scheme for rapid implementation of wrappers." In *Proc. International Conference on Deductive and Object-oriented databases*, 1995.

[11] James M. Purtilo and Joanne M. Atlee. "Module reuse by interface adaptation." *Software–Practice and Experience* 21:6, 1991.

[12] Mary Shaw. "Architectural issues in software reuse: It's not just the functionality, it's the packaging." In *Symposium on Software Reusability*, 1995.

[13] Mary Shaw and Paul Clements. "A field guide to boxology: Preliminary classification of architectural styles for software systems." In *Proc. International Computer Software and Applications Conference*, 1997.

[14] Mary Shaw and David Garlan. *Software architecture: Perspectives on an emerging discipline*, 1996. Prentice-Hall.

[15] Katia Sycara, Keith Decker, Anadeep Pannu, Mike Williamson, and Dajun Zeng. "Distributed intelligent agents." *IEEE Expert* 11:6, 1996.

[16] Tom Wu. "Behind the scenes of the Adventure Web." http://www-tjw.stanford.edu/adventure/impl.html

[17] Daniel M. Yellin and Robert E. Strom. "Interfaces, protocols, and the semi-automatic construction of software adaptors." In *ACM OOPSLA*, 1994.