Procedure Calls Are the Assembly Language of Software Interconnection:

Connectors Deserve First-Class Status

Mary Shaw[1]

January 1994
CMU-CS-94-107
CMU/SEI-94-TR-2
ESC-TR-94-002

School of Computer Science and
Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

Software designers compose systems from components written in some programming language. They regularly describe systems using abstract patterns and sophisticated relations among components. However, the configuration tools at their disposal restrict them to composition mechanisms directly supported by the programming language. To remedy this lack of expressiveness, we must elevate the relations among components to first-class entities of the system, entitled to their own specifications and abstractions.

Keywords: software architecture, system configuration idioms, software design

Presented at Workshop on Studies of Software Design, May 1993.
Also to be published in *Proceedings of Workshop on Studies of Software Design*,
Lecture Notes in Computer Science, Springer-Verlag 1994.

---

[1]This research was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense) and by a grant from Siemens Corporate Research. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies of any of the sponsors.

Table of Contents

Architectural descriptions treat software systems as compositions of components. They focus on the components, leaving the description of interactions among these components implicit, distributed, and difficult to identify. If the interfaces to the components are explicit, they usually consist of import/export lists of procedures and data. Interactions are expressed implicitly through `include` files or `import` and `export` statements, together with the documentation that accompanies various libraries. This view of software architecture organizes information around the components and ignores the significance of interactions and connections among the modules.

This paper begins by discussing the limitations of the conventional approach to system configuration. It then argues that designers must attend as carefully to connections among components as to the components themselves. It closes by proposing a model of system composition in which connectors are first-class entities along with components. Section 1 summarizes current practice and Section 2 describes some of the resulting difficulties. Section 3 gives a fresh view of system configuration and Section 4 sketches a language to support that view.

1. Current practice

When a designer writes a paper about a software system, the first section often includes a diagram and a few paragraphs of text labeled the "software architecture." The text refers informally to common software notions such as pipelines, client-server relations, interpreters, message-passing systems, and event handlers. The diagram usually consists of boxes and lines, but the semantics of the graphic elements varies substantially from one figure to another [5]. Figure 1 is typical of these figures. It depicts a sequence of three processing steps in which the second step also uses four abstract data types and communicates in various ways with a satellite, an interactive workstation, and a database. The components depicted in the diagram may have substructure, but ultimately the implementations of the components must be written in conventional programming languages.

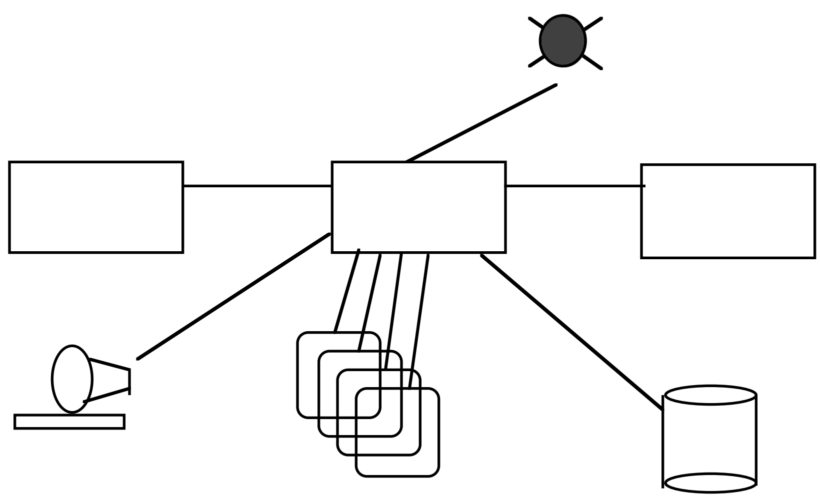

Figure 1: Typical box-and-line depiction of a software architecture

Sometimes components have explicit interface definitions. These define the external structure of the components. They usually consist of lists of procedures, exported data, and perhaps types, exceptions, etc. Ada's `specification` parts and C's `.h` files are examples of such interface definitions. Interfaces do not aggregate these details to reflect the more abstract relations they implement. The specifications of functionality, if any, are generally written in prose; formal specifications that provide details beyond type and signature are relatively rare.

In these conventional designs, all modules have equal status. That is, they are undifferentiated collections of procedures, data, and other constructs of the underlying programming language. Nothing analogous to a type system indicates that a module has special properties, discriminates among different kinds of modules, or identifies specific kinds of analysis available. In the associated implementations, `import` and `export` statements in each module establish the dependencies among modules. In Ada these are `uses` clauses; in C they are `includes`. Specific associations, for example between a procedure definition and its call, rely on matching the names at the definition and use sites.

The models implicit in designers' architectural descriptions (both text and diagrams) do not match the actual realization of these models in code: Architectural models are rich, abstract, spontaneous, and almost wholly informal. However, the implementation languages, including module interconnection languages, are rigorous, precisely defined, and limited in expressiveness to the constructs of the underlying programming language.

As a result of these mismatches, the code fails to capture designers' intentions for the software explicitly and accurately and precise design documentation does not persist into maintenance. This impedes immediate checking and future guidance for development and maintenance activities. Even insofar as the code actually captures parts of the design, it does so in a highly distributed fashion, and it is hard for a reader to get a system-level overview. The need to address abstractions for system configuration is becoming widely recognized [3, 5, 6, 9, 10].

2. Problems with current practice

Current practice in architectural, or system-level, design focuses on components. For a system to work well, however, the relations among components, or *connectors*, require as much design and development attention as the components.

Connectors are less obviously objects of design than are components. After all, the connectors often do not have code—hence identity—of their own. They may be realized in distributed fashion by a variety of system mechanisms. Indeed, system mechanisms such as common scheduling and synchronization policies or the available communications protocols may constrain the designers' choices. Many of the problems with current techniques for architectural definition revolve around inadequacies of the mechanisms for defining component interconnection.

This section reviews some of the problems with the conventional approach of embedding the interactions among components within the components.

2.1. Inability to localize information about interactions

Most current module interconnection techniques, including programming languages such as Ada, depend on import and export commands lodged in the code modules of the system [4, 7]. A link editor then connects the components by matching the names of exported and imported constructs, possibly with guidance from the `import` and `export` statements about the scope of names. This has three major problems

- *Forced agreement in spelling:* The importer must use the same name as the exporter. In at least one case, a "reusable" library was not usable because its names conflicted with existing names of a system.
- *Dispersion of structural information:* An import/export strategy distributes structural information throughout the system. This hides the system structure and impedes reuse by creating embedded references to other components.
- *Forced asymmetry of interaction:* The import/export model assumes asymmetrical relations—there must be an owner and a user, or a master and a slave, or a source and a destination. Although many interactions are binary and asymmetrical, not all are: peer-to-peer communication is symmetrical, client-server relations can have multiple components in each role.

Current practice is also unable to localize the related abstractions. There is no natural home for the definitions that govern a class of interactions. Interactions are provided and defined by the operating system, the programming language, subroutine libraries, embedded languages, and ad hoc user-defined mechanisms. Giving legitimate, uniform status to definitions of interactions would improve system understanding and analysis.

2.2. Poor abstractions

Boxes, lines, and adjacency don't have consistent meaning across system structure diagrams. They usually represent abstract interactions rather than the procedure calls and data declarations of the code. Practical systems have quite sophisticated rules about component interaction and shared representations. Existing definition mechanisms don't allow those design decisions to be captured in the code, so they can't be exploited for analysis or maintenance. The abstractions are hidden for several reasons:

- *Inability to associate related elements and name the cluster:* A module interface may export a large number of named elements. Apart from comments—which have no force—there is no good way to declare that a set of these elements behaves as a coordinated group. Further, there is no way to name the cluster for reference as a whole.
- *Inability to specify relations among related elements:* The ordering and state consistency requirements among a coherent set of calls are usually implicit. This is almost inevitable, for there is no logical place to state them.
- *Inability to specify aggregate properties of a collection of elements:* Even without explicit names, practical systems have quite sophisticated rules about protocols and shared representations. Individual procedure and data element specifications localize information and are not adequate to express these relations.

In Figure 1, shapes help the reader differentiate among different kinds of components, even though the programming language and module interconnection language may not support the distinctions. However, all the connections in that figure are represented in the same way—as simple lines. Figure 2 shows an improved drawing, with different line textures denoting different kinds of interactions.

Designers have abstract, sophisticated intentions for the relations among components. However, they have no reasonable way to capture these design intentions as a permanent part of the software. Even worse, the abstract relations are almost always realized as sequences of procedure calls embedded in modules whose ostensible function is something entirely different. Usual practice does not identify the abstract functions of the procedure calls, nor does it explain the rules about required order of operations.

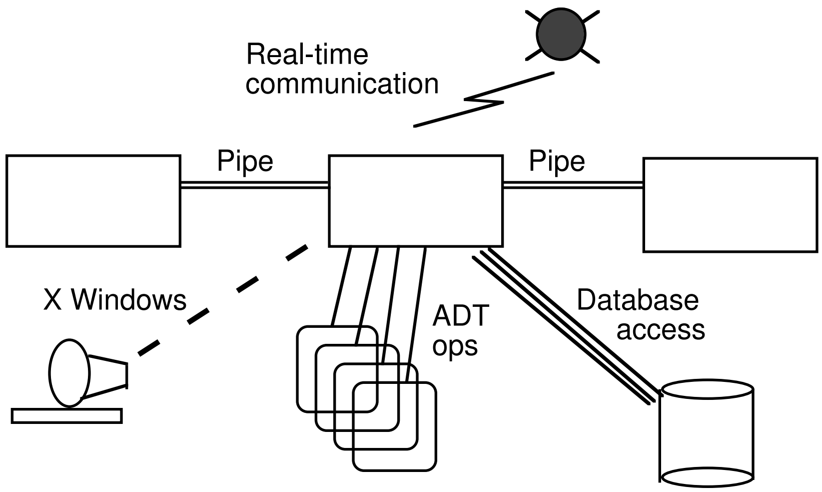

Figure 2: Revised architecture diagram with discrimination among connections

2.3. Lack of structure on interface definitions

As noted above, we lack a widely-used notation for structuring interface definitions so that they cluster coherent subsets of operations. In practice, though, a module is likely to have interfaces for one or more sets of primary users (to provide the overt system function) and additional special operations for such special uses as audit trails, monthly reports, executive control (setpoints or system tuning), system initialization, monitoring, and debugging. Monolithic interfaces can neither clarify nor enforce these distinctions.

Two levels of structure and abstraction are missing:

- *Abstractions for connections:* aggregation of primitive import/exports to show the intended abstract function of the connection.
- *Segmentation of interfaces:* decomposition of an interface into more-or-less conventional segments corresponding to different groups of users or different classes of functionality; each of these may involve several abstract connections.

2.4. Mixed concerns in programming language specification

Programming languages were designed to describe algorithmic operations on data. They are very good at defining data structures and algorithms that operate on those data structures. Extensions allow them to describe computational structures such as concurrency. They are not particularly good at describing reliability, absolute time, and a variety of extra-functional properties. Nor are they good at defining interactions among other modules that are more abstract than procedure calls and shared data.

Two problems result. First, all interactions not directly supported by the programming language must be encoded as sequences of procedure calls. Second, constructs for system composition have been grafted onto programming languages, with less than ideal results (e.g., `private` parts of Ada).

The concerns of architectural design are not with algorithms and data structures, but rather with system topology, assignment of capability to components, interactions among components, and performance characteristics. Therefore it is unrealistic to expect conventional programming languages to serve. Much of the current awkwardness seems to arise from attempts to add capabilities to conventional programming languages that stretch them beyond their design limits.

2.5. Poor support for components with incompatible packaging

When multiple components are reused—for example from different libraries—their interfaces do not always mesh well, even if their computational capabilities are substantially compatible. For example,

- If a component is cast as a filter, it can't be used as procedure because it does I/O to pipes instead of through procedure parameters.
- If a component is cast interactively, it often can't be called by another program.
- If semantics are suitable but packaging details such as name and parameter order differ, the user must write ad hoc conversions.

Something akin to typing is going on here: to use a component, you need to know not only what it computes but also how it delivers that computation. In many of these cases, the incompatibilities can be overcome by introducing mediators that accommodate discrepancies between the protocol expected by the component and the protocol requested by the designer.

2.6. Poor support for multi-language or multi-paradigm systems

The connection between components is substantially independent of the programming languages of the components. For example, this is usually the case when the components run as separate processes. Connectors that work naturally in these cases include unix pipes and many message systems.

In other cases, the connection between components depends directly on the programming language. This is often the case when components share assumptions about runtime systems such as representations of data types.

The conditions under which components in different languages can interact musts be detailed in such a way that a system development tool can tell which connections are allowable, which can be mediated, and which cannot be supported.

Furthermore, tools intended to support one architectural paradigm—object management tools, unix shell—offer little assistance in creating a system that mixes different architectural idioms.

2.7. Poor support for legacy systems

Most software development now involves modification of existing systems. Most of these systems evolved without configuration tools any more sophisticated than, say, `make`.

Syntactic tools make it possible to extract the signatures—the names and types of imported and exported entities. However, they offer no help in recovering the higher-level intentions such as which set of procedures collectively implements a given abstract protocol. Over half of system maintenance effort goes into deciphering what the software already does, so the inability to record and retain the designer's higher-level intentions about component interactions is a major cost generator.

3. A fresh view of software system composition

Systems are composed from identifiable components of various distinct types. The components interact in identifiable, distinct ways. *Components* roughly correspond to compilation units of conventional programming languages. *Connectors* mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary mechanism required. Connectors do not in general correspond individually to compilation units; they manifest themselves as table entries, instructions to a linker, dynamic data structures, system calls, initialization parameters, servers that support multiple independent connections, and the like.

It is helpful to think of the connector as defining a set of *roles* that specific named entities of the components must *play*.

Software systems thus comprise two kinds of distinct, identifiable entities: *components* and *connectors*.

- *Components* are the locus of computation and state. Each component has an *interface specification* that defines its properties. These properties include the signatures and functionality of its resources together with global relations, performance properties, etc. Each is of some type or subtype (e.g., filter, memory, server). The specific named entities visible in the interface of a component are its *players*.
- *Connectors* are the locus of relations among components. They mediate interactions but are not "things" to be hooked up (they are, rather, the hookers-up). Each connector has a *protocol specification* that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc. Each is of some type or subtype (e.g., remote procedure call, pipeline, broadcast, event). The specific named entities visible in the protocol of a connector are *roles* to be satisfied (e.g., client, server).

Components may be either primitive or composite. Primitive components are usually code in the conventional programming language of your choice. Composite components define configurations in a notation independent of conventional programming languages. This notation must be able to identify the constituent components and connectors, match the connection points of components with roles of connectors, and check that the resulting compositions satisfy the specifications of both the components' interfaces and the connectors' protocols.

Similarly, connectors may be either primitive or composite. They are of many different kinds: shared data representations, remote procedure calls, data flow, document exchange standards, standardized network protocols. The set is rich enough to require a taxonomy to show relations among similar kinds of connectors. Primitive connectors may be implemented in a number of ways: as built-in mechanisms of programming languages (e.g., procedure calls associated by a linker); as system functions of the operating system (e.g., certain kinds of message passing); as library code in conventional programming languages (e.g., X/Motif); as shared data (e.g., Fortran `COMMON` or Jovial `COMPOOL`); as entries in task or routing tables; as a combination of library procedures and a single independent process for the connector (e.g., certain kinds of communication services); as interchange formats for static data (e.g., RTF); as initialization parameters (e.g., process priority in a real-time operating system) and probably in a variety of other ways. Composites may also appear in these diverse forms; we need (but do not yet have) ways to define them, as well.

Connectors are properly treated separately from components because:

- *Connectors may be quite sophisticated*, requiring elaborate definitions and complex specifications that deserve their own homes. In many cases, no single component is the appropriate location for a protocol specification.
- *The definition of a connector should be localized.* Just as good methodology requires a single location for the definition of a component, good methodology requires a single location for the definition of an interaction. This supports both design (especially analysis during design) and maintenance. Further, connectors can be rich enough for their definitions to deserve their own homes.
- *Some information about the system does not have a proper home in any component.* For example, in a real-time system it may be appropriate for tasks to declare their needs and for a separate scheduler to satisfy them.
- *Connectors are potentially abstract.* They may be parameterizable. They may define classes of interactions that require additional scripting at the time of instantiation. Users may wish to define their own connectors, to make their own specializations, of existing connectors, or to compose their own connectors. A single connector may be instantiated multiple times in a single system; for example, a multicast capability could support many distinct sets of communicating processes.
- *Connectors may require distributed system support:* The mechanism required by a connector is not always localized to individual uses. For example, a message passing system may require exactly one server per processor for any number of communicating processes.
- *Components should be independent*. The interface specification of a component should provide a complete specification of the capabilities of that component but remain silent on how it is actually used.
- *Connectors should be independent.* A single (high-level) connector might mediate relations for a dynamically changing set of components. Wiederhold describes such a scheme [11].
- *Relations among components are not fixed.* A component may be capable of being used differently by different kinds of connectors. For example, a client might be indifferent to whether its server is dedicated, shared, or distributed. In addition, system connectivity can change dynamically.

System compositions quite frequently reuse patterns of composition; some of these patterns are commonly understood, at least intuitively: pipe/filter, client/server, layered system, blackboard. These common idioms can be defined as generic patterns that restrict the types of components and connectors to be used and describe how the pattern is implemented [5]. This may involve constraining the topologies of interconnection. Current module interconnection languages are wholly inadequate to this task for reasons elaborated in Section 2.

4. An architectural language with first-class connectors

Software system composition involves different tasks from writing modules: the system designer defines roles and relationships rather than algorithms and data structures. These concerns are sufficiently different to require separate languages. The architectural language must support system configuration, independence of entities (hence reusability), abstraction, and analysis of properties ranging from functionality to security and reliability [9]. The design of such a language is not straightforward. In addition to having a syntax, it must:

- Define semantics for connectors and their compositions.
- Generalize the import/export rules to address asymmetry, multiplicity, locality, abstraction, and naming.
- Establish type structures for system organizations, components, connectors, and the primitive units of association of these elements; this includes defining taxonomies for the types.
- Set out appropriate rules for architectural abstractions.

This section contains some initial notes on these language design problems.

4.1. Language structure

As suggested above, the language needs separate (but parallel) constructs for components and connectors. It must provide notations for composition and a set of primitives (including primitives defined in conventional programming languages). For simplicity, the constructs for components and connectors can be similar. Figure 3 suggests the essential character of the language. Each construct is typed. It has a specification part and an implementation part. The specification part defines specific units of association to be used in system composition.

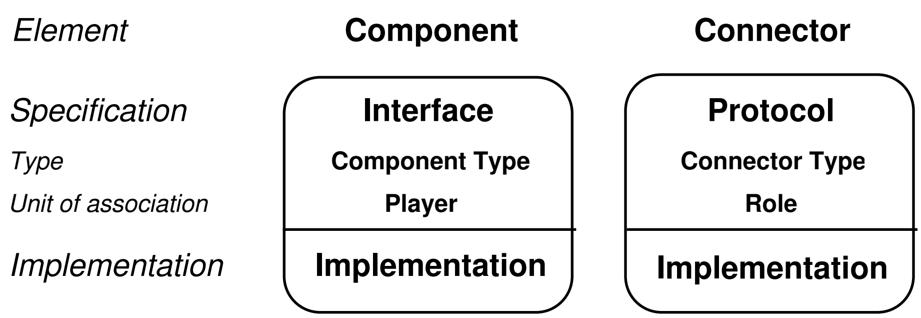

Figure 3: Gross structure of an architecture language

It is sometimes useful to say explicitly that an element is primitive; this means that it is not further defined at the architecture level but is implemented in a programming language or with system-level mechanisms.

For a nonprimitive element, the implementation part consists of a parts list, composition instructions, and related specifications. This establishes explicit associations and specification matches, thereby breaking free of name matching as the sole means of making connections.

The specifications should be "open" with respect to construction and analysis tools. Many different approaches are available for specifying and verifying system properties of interest; the languages should be able to accept those as uninterpreted expressions and interact appropriately with the specialized tools.

4.2. Connectors and their semantics

Most programming languages support some sort of intermodule connection. It usually supports only the primitive units of association of the language, such as procedure calls. Making the connectors first-class requires careful analysis of all the roles these constructs play in the definition of a system.

A connector mediates the interaction of two or more components. It is not in general implemented as a single unit of code to be composed. The previous section describes a number of the implementation possibilities. Whatever the implementation of a connector (especially an abstract one), detail about the implementation technique is encapsulated when the connector is used. Moreover, many or all connectors of the same type may share the same code or data. Allen is investigating formal specifications of the semantics of certain classes of connectors [1].

Like components, connectors require specifications. Specifications for connectors are called *protocols*. Since protocols can be of many different kinds, languages should allow for flexible specifications. One possibility is heavy use of property lists, with some standard attributes and some attributes specific to particular connector types. This allows for properties as diverse as:

- guarantees about delivery of packets in a communication system
- ordering restrictions on events using traces or path expressions
- incremental production/consumption rules about pipelines
- distinguishing between the roles of clients and servers
- parameter matching and binding rules for conventional procedure calls
- restrictions on parameter types that can be used for remote procedure calls

Primitive connectors include at least the ones directly supported by the programming language or operating system. These certainly include the procedure call and data accessors of each programming language; they also include language-specific process interactions such as the Ada `rendezvous`. Careful attention to the roles involved in primitive connectors show the need to support asymmetry: a procedure has a definer and multiple callers; data has an owner and multiple users. On the other hand, in certain classes of event systems all entities are equally entitled to generate and recognize events, so it is also necessary to define symmetric roles in a protocol. The usual import/export or provides/requires relation is too restrictive.

The simplest kind of abstract connector is binary (its protocol has two roles, for example definer and user). Some of these are direct analogs of the language-supported connectors, such as procedure call. At the architecture level the relation is often more abstract. For example, it may be desirable to separate from the definition of a procedure the decision about whether it is to be a local or remote procedure call.

N-ary connectors that involve multiple components are also important. These may be symmetrical, with all connected components playing the same role (e.g., multicast). They may (probably more commonly) be asymmetrical, with different roles for different components or sets of components (e.g., client-server systems).

Connectors are often implemented as sets of procedures. A set of procedures frequently has an associated set of rules or assumptions about how the procedures will be used. These rules are often highly implicit. They may restrict the order in which procedures are called or require relations among parameter values. These rules amount to protocols for the interaction. For example, the operations of an abstract data type are used as a bundle; they often have order restrictions such as "initialize must be called before anything else; push must be called at least as often as pop." Explicit restrictions may be expressed in various ways, as path expressions or traces for execution order restrictions. Although it's not conventional, it is useful to think of abstract data types as having a protocol that guides the use of the operations. This not only captures essentials such as execution order restrictions but also decouples the selection of the abstract type from the selection of an implementation. This is not unlike Larch's separation between abstract properties and actual implementations.

Figure 4 suggests the protocols required to construct the system of Figures 1 and 2. These protocols should exist as independent definitions in the computing environment. They may take parameters (including partial specifications) and may support several variants. When they are used, additional specifications may be needed to specialize the protocol or select a particular form.

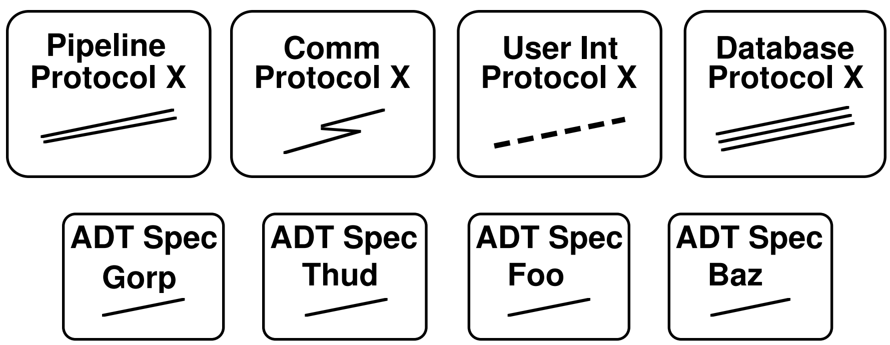

Figure 4: Constellation of protocol specifications required by example

Figure 5 shows some of the information that should be in the interface of the central component of Figures 1 and 2. This syntax is suggestive rather than definitive. Annotations on the left side show correspondences to the line styles of the diagram. Each of the nine lines of the interface describes an interaction with some other component.

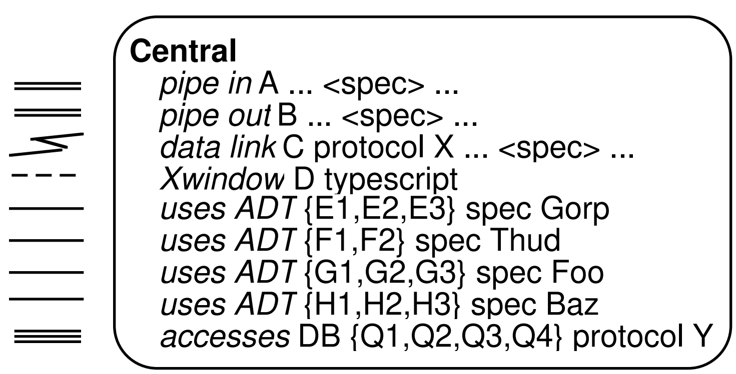

Figure 5: Interface specification of central component, referring to protocols

Note that in several cases the normal notion of exporting some resources and importing others does not apply well. For the pipes, additional specifications limit the type of information passing through the pipe. Similarly, a communication protocol may require additional specifications. The four abstract data types are all of the same general category of protocols; the bracketed names are the names by which the central component will call designated operations of the four types.

4.3. Architectural type structures

A problem akin to type checking arises at three points in an architectural language. Two appear in the preceding discussion: the types of components and of connectors. As with any type system, these express the designer's intent about how to use the element properly. To be useful, they must also have some enforcement power.

Architectural types describe expected capabilities. They can limit the legitimate ways to use the construct. They can abbreviate restrictions on what can appear in the construct's specification. Examination of real systems shows that type hierarchies of this sort are useful. For example, there are many kinds of memories (components) and many kinds of event systems (connectors). Defining type structures for these elements requires creation of taxonomies that catalog and structure the type variations.

The third place where something like type checking shows up is at the actual point of associating *players* of components with *roles* of connectors. Each of the named entities in the interfaces (of components) and protocols (of connectors) must have enough type and other specification to check on whether the connector definition allows the components to be associated as requested.

Because of the need to reuse components and connectors in settings that aren't all quite alike, it is important to deal reasonably with associations that don't quite match. A very common example is the use of a unix pipe to send data to a file. The definition of a unix filter will probably say it's intended to interact with other filters through pipes. However, it is often (but not always) well-defined to substitute a file (passive component) for a filter (active component). The language must provide a way to define and control possible fix-ups, for example by supporting rules of the flavor "this pipe will accept a file in the role of filter under the following circumstances ...". Some of the interesting alternatives include:

- Associate anyhow: it will work without extra effort (some subtype relations).
- Rearrange or reformat information (data re-formatters [2], parameter re-mappers [8]).
- Wrap the component in a converter (a procedure wrapper for a filter would feed the input parameters to the input pipe of the filter and collect the result from the output pipe for delivery as a single value).
- Convert data to and from a shared form (interchange format).
- Convert data of one component to the form expected by another (pairwise compatibility; common message format, but data may need to be interpreted).
- Insert conversion module (buffer).
- Just say no.

The history of type coercion in conventional programming languages (especially PL/I) provides convincing evidence that this capability must remain firmly under control of the software designer at all times.

A special case of compatibility checking and enforcement arises when components are written in different programming languages. The difficulty of accomplishing this depends on the extent of the shared assumptions between the components.

- Sometimes there is no problem: One common easy case when two languages share runtime systems with common runtime representations, procedures, and protocols (Fortran/Snobol). A second common case is explicit, loosely-coupled interactions (unix pipes with ASCII streams).
- Sometimes the problem can be resolved with mediation as described above.
- Sometimes an external representation standard (RTF, PICT, SYLK) or an inter-language procedure call can serve as a cross-language connector.
- Sometimes it's simply too hard (languages with essentially different assumptions: rule-based; static imperative; dynamic).

4.4. Abstractions for higher-level connectors

The discussion so far has mentioned many different higher-level connectors. These include client/server relations, messages, event handlers, multicast communication, radio communication links, unix pipes, shared data, interaction through X/Motif or SQL scripts, hierarchical layers, and blackboards. The example of Figures 1 and 2 might be instantiated with SQL, X/Motif, various data abstractions with usage restrictions, unix pipes, and radio data links. The software development environment should provide the most common of these, either as part of the programming language, as basic operating system capability, or as part of the infrastructure (SQL, X/Motif). Protocols for this baseline collection should be primitive to the architectural language. It is still unclear exactly how to define the association of procedure calls with abstract protocols precisely, and the semantics of abstract connectors are also an open question at this time. However, it is clear that connectors have interesting internal structure, much as unix pipes contain buffers.

As discussed in section 2.3, component interfaces often have several distinct segments in order to establish different kinds of relations. Often these will have corresponding protocols.

An architectural language must support not only individual abstract connectors, but also high-level compositions that involve a number of connectors in specific relations to one another. For example, the language must be capable of capturing the high-level architectural idioms such as blackboards, interpreters, and various domain-specific architectures as abstractions. I conjecture that non-primitive connectors are the appropriate way to do this, but it isn't yet demonstrated.

5. The promise of explicit architectural notations

What makes the construction of composable systems different from conventional programming? First, composing a system from subsystems is unlike programming the algorithms and data structures that lie within the primitive subsystems. The semantics of the components, the locality of reasoning, the character of interaction with other components, the properties of interest, and the nature of the reasoning are all different. Second, we are liberating ourselves from thinking of the task as merely "programming". We are not merely building a program that receives inputs, executes, and terminates—we are building a system that has an enduring existence in some larger environment. Third, our units of manipulation are not simply conventional modules (which might export data, procedures, and perhaps tasks), but rather components and connectors.

Identifying connectors as first-class entities in a system can help to break us out of the programming-language mindblock for system composition languages. Legitimizing higher-level interactions among components allows us to understand the procedure call as one—perhaps the primary—primitive connector of pairs of modules. More significantly, it allows us to recognize higher-level connectors as critical to system design. We must learn to support higher-level connectors—composite components and connectors whose properties, expressed in their interface and protocol specifications, are as understandable as their constituents.

The view proposed in Sections 3 and 4 addresses the problems identified in Section 2.

- It specifically provides for localizing information about interactions: Nonprimitive components can invoke rich relations, and they concentrate structural information.
- It introduces abstractions for interactions and provides a starting point for user-defined abstractions and aggregations.
- It partially addresses the interface structure problem by using the *roles* of connectors to identify related operations as *players*.
- It separates architectural concerns from programming concerns by providing different language constructs with different semantics.
- It makes provisions for a type-checking system that can adapt to mild mismatches, thereby enhancing opportunities for reuse.
- It clarifies the conditions under which programming languages can be mixed.
- It offers prospects for improved support of legacy systems by making the architectural design of the system explicit.

The principled use of compositional structures should have a dramatic effect on software production. It should permit the choice of design paradigms to match desired system characteristics. It should allow the development of application-specific frameworks and reference architectures. It should provide the basis for exploiting compositional properties of systems for formal analysis, code generation, and software reuse. It should support a high level of visible abstraction for systems designers so that large systems can be more easily designed, understood, maintained and enhanced. It should enable us to better accommodate old code by providing ways to recover partial architectural information from existing systems.

Acknowledgments

These ideas have been evolving over a period of several years. They have been substantially sharpened and worked out in discussion with David Garlan over the past two years. Daniel Klein implemented the first prototype of the base language, Greg Zelesnik is extending the prototype, and Rob DeLine's critique stimulated clarification of the terminology. The CMU Software Architecture Reading Group has provided valuable feedback both on drafts of the paper and on the underlying ideas.

References

[1] Robert Allen and David Garlan. *Formalizing Architectural Connection. Proc. Sixteenth International Conference on Software Engineering*, 1994.

[2] Brian Beach. Connecting Software Components with Declarative Glue. *Proc. Fourteenth International Conference on Software Engineering*, 1992.

[3] Barry W. Boehm and William L. Scherlis. Megaprogramming. Proc. DARPA Software Technology Conference 1992, pp. 63-82.

[4] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus Programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80-86, June 1976.

[5] David Garlan and Mary Shaw. *An Introduction to Software Architecture*. In V. Ambriola and G. Tortora (eds), Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

[6] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, vol 17, no 4, October 1992, pp. 40-52.

[7] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software* vol 6, no 4, November 1986, pp. 307-334.

[8] James Purtilo and Joanne Atlee. Module Reuse by Interface Adaptation. *Software: Practice and Experience*, 21(6): 539-556, June 1991.

[9] Mary Shaw and David Garlan. *Characteristics of Higher-Level Languages for Software Architecture*. Unpublished manuscript, 1993.

[10] Gio Wiederhold, Peter Wegner, and Stefano Ceri. *Toward Megaprogramming*. Stanford University Technical Report STAN-CS-90-1341, 1990.

[11] Gio Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3):38-49, March 1992.

Change History

Fixed up TR format 12/22/93, afternoon

Convert to TR format 12/21/93, morning

Fix refs Dec 20, afternoon; sent LNCS version to David Lamb

Tighten prose, Dec 17, evening

Clean up figures Dec 16, 1993 afternoon

Acitvate passive voice Dec 16, 1993 morning

Tighten early prose Dec 15, 1993

Purge obsolete vocabulary; rewrite locally for sensibility. Dec 6, 1993.

Convert to Lamb format Oct 31, 1993

Add text for some finish-up notes; make quote marks pretty; remove Steamboat notes 8:40 AM March 31, 1993

Finish rewrite 11:20 PM March 30, 1993

Convert to David Lamb's format 3:30 PM March 30, 1993

Overall fresh pass on prose 9:30 PM March 29, 1993

Added gobs of notes from DSSA/Prototech meeting 9:28 AM Jan 14, 1993

Finish rewrite, spell check 11:23 PM Jan 7, 1993

Extensive rewrite of second half, with new figures 6:15 PM Jan 7, 1993

Continue smoothing and fleshing out prose 8:55 AM Jan 7, 1993

Rework prose, reformat, resize pictures 9:47 PM Jan 6, 1993

Converted to Word 8:15 AM Sep 1 1992

Rework language elements section 8:10 AM Sep 1, 1992

Pick up characterization from msg to Jeannette; flesh out problems 1:20 AM Sep 1, 1992

Add detail, figures 7:30 PM Aug 31, 1992

Restate point of view 5:30 PM Aug 30, 1992

Emphasized abstract protocols, spell ck 3:20 PM Aug 16, 1992

Expanded outline, started prose conversion 11:53 AM Aug 16, 1992

Expand outline and add detail 11:34 AM Aug 15, 1992

Add Flesh 10:15 AM Aug 14, 1992

Created 9:00 AM Aug 12, 1992