Characteristics of Higher-level Languages for Software Architecture

Mary Shaw David Garlan

December 1994

CMU-CS-94-210

Also appears as CMU Software Engineering Institute Technical Report
CMU/SEI-94-TR-23 ESC-TR-94-023

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

© David Garlan and Mary Shaw

The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grants CCR-9109469 and CCR-9112880; and by a grant from Siemens Corporate Research; this work was created in the performance of Federal Government Contract Number F19628-90-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a Federally Funded Research and Development Center. The Government of the United States has a royalty-free government purpose license to use, duplicate or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, the National Science Foundation, Siemens Corporation, or Carnegie Mellon University. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

Keywords: software architecture, architecture description languages, architecture representation languages, software design, module interconnection

Abstract

As the size and complexity of software systems increases, the design and specification of overall system structure—or software architecture—emerges as a central concern. Architectural issues include the gross organization of the system, protocols for communication and data access, assignment of functionality to design elements, and selection among design alternatives.

Currently system designers have at their disposal two primary ways of defining software architecture: they can use the modularization facilities of existing programming languages and module interconnection languages; or they can describe their designs using informal diagrams and idiomatic phrases (such as "client-server organization").

In this paper we explain why neither alternative is adequate. We consider the nature of architectural description as it is performed informally by systems designers. Then we show that regularities in these descriptions can form the basis for architectural description languages. Next we identify specific properties that such languages should have. Finally, we illustrate how current notations fail to satisfy those properties.

1 Introduction

As the size and complexity of software systems increase, the design and specification of overall system structure become a more significant issue than the choice of algorithms and data structures of computation [DK76, Sha89]. Structural issues include the gross organization of the system; the global control structure; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; scaling and performance; and selection among design alternatives. This is the *software architecture* level of design [GS93b, PW92].

Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components. Such a system may in turn be used as a (composite) element in a larger system design.

The use of software architectures is pervasive in the informal diagrams and idioms that people use to describe system designs. Designers typically draw "architectural" diagrams consisting of boxes and connecting lines, and allude to common paradigms for describing their meaning. For example, the relation between entities might be described as an instance of a "client-server model," a "pipeline," or a "layered architecture."

Unfortunately, diagrams and descriptions such as these are highly ambiguous. At best they rely on common intuitions and past experience to have any meaning at all. Moreover, system designers generally lack adequate concepts, tools, and decision criteria for selecting and describing system structures that fit the problem at hand. It is virtually impossible to answer with any precision the many questions that arise during system design. What is a "pipeline" architecture, and when should one pick it over, say, a "layered" architecture? What are the consequences of choosing one structural decomposition over another? Which architectures can be composed with others? How are implementation choices related to the overall performance of these architectures? And so on.

The problem of describing structural decompositions more precisely has traditionally been addressed by modularization facilities of programming languages and module interconnection languages [PDN86]. These notations typically allow an implementor to describe software system structure in terms of definition/use or import/export relationships between program units. This supports many features for programming-in-the-large, such as separate compilation, well-defined module interfaces, and module libraries.

However, as we show later, such language support is inadequate for architectural descriptions. In particular, descriptions at the level of programming language modules provide only a low-level view of interconnections between components, in which the only directly expressible relationships between components are those provided by the programming language. Moreover, they fail to provide a clean separation of concerns between architectural level issues and those related to choice of algorithms and data structures.

More recently, a number of component-based languages have been proposed and implemented. These languages describe systems as configurations of modules that interact in specific, predetermined ways (such as remote procedure call, messages, or events [Pur88, Kra90, Pou89, Mak92, Bea92]) or enforce specialized patterns of organization [D^+91, Ros85, L^+88]. While such languages provide new ways of describing interactions between components in a large system, they too are typically oriented around a small, fixed set of communication paradigms and programming-level descriptions or they enforce a very specialized single-purpose organization. This makes them inappropriate for expressing a broad range of architectural designs.

In this paper we consider the need for new higher-level languages specifically oriented to the problem of describing software architecture. First we show how ideas from "classical" language design apply to the task of describing software architectures. We then detail the characteristics such languages should have in the areas of composition, abstraction, reusability, configuration, heterogeneity, and analysis. Finally, we show how existing approaches fail to satisfy these properties, thus motivating the need for new language design. In taking this general point of view, the intention is not to propose a particular language—indeed, we believe that no single language will be sufficient for all aspects of architectural description—but rather to establish the framework within which architectural language design must take place.

2 The Linguistic Character of Architectural Description

We now illustrate how the structure of the architectural task is amenable to treatment as a language problem and argue that the principles learned from the design of programming languages can serve us well in designing notations for software architecture. The argument is as follows:

- Analysis of commonly-used architectures reveals common patterns, or idiomatic constructs.
- Those constructs rely on a shared set of common kinds of elements; similarly, they rely on a shared set of common intermodule connection strategies.
- Languages serve precisely the purpose of describing complex relations among primitive elements and combinations thereof.
- It makes sense to define a language when you can identify appropriate semantic constructs; we find an appropriate basis in the descriptions of architectures.

Common patterns of software organization

Papers describing software systems often dedicate a section to the architecture of the system. This section typically contains a box-and-line diagram; usually boxes depict major components and lines depict some communication, control, or data relation among the components. The boxes and lines mean different things from one paper to another, and the terms in the prose descriptions often lack precise meaning. Nevertheless, important ideas are communicated by these descriptions.

Some of the informal terms refer to common, or idiomatic, patterns used to organize the overall system. These are often widely used among software engineers in high-level descriptions of system designs. A number of the more pervasive patterns have been identified in descriptions of architectural idioms [GKN88, GS93b, Sha89, Sha91], and material based on these patterns is beginning to appear in courses on architectural design of software [GSO+92].

Among the more common architectural patterns are:

Pipes and filters Graph of incremental stream transformers.
Examples: Unix pipes, signal processing.

Client-server Shared services provided by request to distributed clients.
Examples: File servers, distributed databases.

Hierarchical layers System partitioned into layers, which act as virtual machines.
Examples: OS kernels, ISO OSI.

Communicating processes
System is composition of independent, concurrent processes.

	Examples: Many distributed systems.
Interpreter	Execution engine bridge gap between the abstract programs and the machine on which they must run. Examples: Rule-based systems, blackboard shell.

Software developers would clearly benefit from having more precise definitions of these structures, including the forms in which they appear and the classes of functionality and interaction they provide. Initial steps toward this goal have recently appeared [GD90, GN91].

Common components and interconnections

In the diagrams of architectural descriptions, the boxes usually have labels that are highly specific to the particular system: "lexical analyzer," "alias table," "requisition slip datafile." The lines (or sometimes adjacencies) that represent interactions are similarly specific: "identifiers," "update requests," "inventory levels."

Examination of these descriptions shows that if the specific functionality of the elements (boxes) is set aside, the remaining structural properties often fall into identifiable classes. For example, here are some of the classes of components that appear regularly in architectural descriptions:

(Pure) computation	Simple input/output relations, no retained state. Examples: math functions, filters, transforms.
Memory	Shared collection of persistent structured data. Examples: database, file system, symbol table, hypertext.
Manager	State and closely related operations. Examples: abstract data type, many servers.
Controller	Governs time sequences of others' events. Examples: scheduler, synchronizer.
Link	Transmits information between entities. Examples: communication link, user interface.

The interactions among components are also of identifiable kinds. Some of the most common are:

Procedure call	Single thread of control passes among definitions. Examples: ordinary procedure call (single name space), remote procedure call (separate name spaces).
Data flow	Independent processes interact through streams of data; availability of data yields control. Examples: Unix pipes.
Implicit triggering	Computation is invoked by the occurrence of an event; no explicit interactions among processes. Examples: event systems, automatic garbage collection.
Message passing	Independent processes interact by explicit, discrete handoff of data; may be synchronous or asynchronous. Examples: TCP/IP.
Shared data	Components operate concurrently (probably with provisions for atomicity) on same data space Examples: blackboard systems, multiuser databases.

Instantiation	Instantiator uses capabilities of instantiated definition by providing space for state required by instance. Examples: use of abstract data types.

The significant thing about these classes of components and forms of interaction is that they are shared by many different architectural idioms—that is, the higher-level idioms are composed from a common set of primitives.

Critical elements of a design language

Programming language design of the 1970s taught us that a language requires

Components	Primitive semantic elements and their values
Operators	Functions that combine components
Abstraction	Rule for naming expressions of components and operators
Closure	Rule to determine which abstractions can be added to the classes of primitive components and operators
Specification	Association of semantics to the syntactic forms

For conventional programming languages, which deal with algorithms and data structures, the components include integers, floating point numbers, strings, records, arrays, etc. The operators include iteration and conditional constructs and type-specific operators such as +, -, *, /. Abstraction rules allow definition of macros and procedures. The closure rules of some languages, but not of others, make procedures first-class entities; in a data abstraction language user-defined types are supposed to be ratified by the closure rule, but this is usually incompletely carried out. The specification of semantics may be either formal or informal (e.g., the reference manual).

The language problem for software architecture

Software architectures deal with the gross allocation of function to components in the system, with data and communication connectivity, and with overall performance and system balance. These are very different from the concerns of conventional programming languages. As a result, the specific forms of the various language elements are also different:

Components	Module-level elements (possibly but not necessarily compilation units of a conventional programming language); kinds of components suggested above; kinds of components can be characterized and used in many different ways.
Operators	Interconnection mechanisms as suggested above; most current systems now support only procedure call and perhaps shared data.
Patterns	Compositions (e.g., "client-server relation") in which code elements are connected in a particular way.
Closure	Conditions in which composition can serve as a subsystem in development of larger systems.

Specification Not only of functionality, but also of performance, fault-tolerance, etc.

These identifications of architectural components and techniques for combining them into subsystems and systems provide the basis for designing languages for architectural description. As we now detail, such a language would support not only simple expressions defining connections among simple modules but also subsystems, configuration of subsystems into systems, and common paradigms for such combinations.

3 Desiderata for Architectural Description Languages

The broad outlines of a system to support architectural design are relatively clear from informal experience. Such a system must provide models, notations, and tools to describe architectural components and their interactions; it must handle large-scale, high-level designs; it must support the adaptation of these designs to specific implementations; it must support user-defined or application-specific abstractions; and it must support the principled selection of architectural paradigms.

In such a system there is clearly a close interplay between language and environment: a language is necessary to have precise descriptions, while an environment is necessary to make those descriptions usable and reusable. However, focusing primarily on the linguistic issues, we can elaborate six classes of properties that characterize what an ideal architectural description language would provide: composition, abstraction, reusability, configuration, heterogeneity, and analysis.

3.1 Composition

It should be possible to describe a system as a composition of independent components and connections.

Composition capabilities allow independent architectural elements to be combined into larger systems. This has three important aspects: First, an architectural language must allow a designer to divide a complex system hierarchically into smaller, more manageable parts, and conversely, to assemble a large system from its constituent elements. Second, the elements must be sufficiently independent that they can be understood in isolation from the system in which they are eventually used. Third, it should be possible to separate concerns of implementation level issues (such as choice of algorithms and data structures) from those of architectural structure.

The need to handle large-scale systems at suitably high levels of abstraction implies both that an architectural description must be modular and that new system elements can be created by combining existing ones. Modularity is required to factor a complex description into smaller conceptual units. The language's closure rule must allow entities of an architectural description to be viewed as primitive at one level of description and as composite structures at a lower level of description. The need for independence of architectural elements—both components and connections—follows from the desire to represent these elements as standalone definitions. This not only makes it possible to reason about an architectural description in terms of its constituent parts, but also avoids preempting the use of those parts in many different contexts.

Needs for composition are evident in the use of common architectural idioms. For example, a pipeline architecture is modularized as a sequence of pipes and filters while a layered architecture is modularized as a collection of abstraction layers that interact according to established rules. Moreover, a filter in a pipeline architecture might itself be internally represented as another pipeline system or even as an instance of a completely different architecture. A pipe, which can be viewed abstractly as a simple connection of a pipeline architecture, might itself be internally represented as complex architecture such as a layered protocol. Independence of pipes and filters allows us to understand a given instance of the architecture in terms of the functional composition of the elements. Independence also allows us to use pipes and filters in contexts other than strict pipeline architectures. For example, a filter might be used in any system that requires a data stream transformation. Similarly, a pipe might be used anywhere that data transmission is required—for example, between a console and a command interpreter in a user interface system.

3.2 Abstraction

It should be possible to describe the components and their interactions of a software architecture in a way that clearly and explicitly prescribes their abstract roles in a system.

Abstraction addresses the need to describe design elements at a level that matches the intuitions of designers. It is not sufficient to provide low-level mechanisms into which those intuitions can be translated. In particular, it should be possible to represent as first class abstractions new architectural patterns and new forms of interaction between architectural elements.

In all software systems, abstraction is used to suppress unnecessary detail yet reveal important properties. For example, high-level imperative programming abstractions suppress issues such as register usage but reveal sequential control flow abstractions (loops, exceptions, etc.). Similarly, programming language module interfaces suppress issues of implementation, but may reveal definition/use dependencies.

The architectural level of design requires a different form of abstraction to reveal high-level structure so that the distinct roles of each element in the structure are clear. Such a description is needed to make explicit the kind of architectural elements that are being used and the relationships between those elements. It is not enough, for example, to describe a system simply as a set of modules whose interfaces are collections of procedure calls, because the structural relationships between modules is encoded in the low-level details of procedure semantics and definition/use dependencies.

For example, architectural abstractions should be capable of explicitly indicating that components are related by a client-server relationship. While both the client and the server may be implemented as a traditional module, the "client-server-ness" of the architecture is not revealed simply by looking at the procedure call interfaces. In a similar way, it should be possible to describe a system of pipes and filters without having to rely on implicit coding conventions, or unstated assumptions about the operating environment (e.g., that two processes will communicate through the Unix pipes).

3.3 Reusability

It should be possible to reuse components, connectors, and architectural patterns in different architectural descriptions, even if they were developed outside the context of the architectural system.

While many languages permit reuse of individual components, few make it possible to describe generic patterns of components and connectors. Such patterns, or frameworks, should permit one to describe a family of system architectures as an open-ended collection of architectural elements, together with constraints on their structure and semantics. This form of reuse differs from the reuse of components from libraries. Libraries supply complete closed or parameterized components whose identities are retained in the final systems. Architectural patterns, however, require further instantiation of substructure and indefinite replication of relations. That is, component libraries supply leaves of the "is-composed-of" structure of a system, whereas architectural patterns supply sets of internal nodes.

Systems are rarely conceived in isolation; they are usually instances of a family of similar systems that share many architectural properties. These shared properties may be structural—such as a specific topology of components and connectors. Or they may simply represent constraints on the use of certain kinds of architectural elements, without defining how they should be connected. An architectural language must permit the characterization of these shared properties.

Needs for reusability go considerably beyond the capabilities of most module constructs provided by programming languages. Typically such modules can be parameterized, as with Ada generics or SML functors. But few languages allow one to talk about parameterized *collections* of modules, or structural patterns. As a simple example, consider a pipe-line architecture, which uses pipes and filters as its basic architectural elements, but also constrains the topology to be a linear sequence.

3.4 Configuration

> *Architectural descriptions should localize the description of system structure, independent of the elements being structured. They should also support dynamic reconfiguration.*

Properties of configuration permit the architectural structure of a system to be understood and changed without having to examine each of the individual components in a system. A consequence is that a language for architectural description should separate the description of composite structures from the elements in those compositions. This allows reasoning about the composition as a whole.

Dynamic reconfiguration is needed to allow architectures to evolve during the execution of a system. This reflects common practice in which new components can be created by other components, and new interactions between components can be initiated. In an object-oriented architecture, for example it is essential to be able to create new objects; in a system of communicating processes it is often essential for processes to be created and killed.

3.5 Heterogeneity

> *It should be possible to combine multiple, heterogeneous architectural descriptions.*

There are two distinct aspects of heterogeneity. The first concerns the ability to combine different architectural patterns in a single system. For example, it should be possible to define a single component that communicates with some components through a pipe, but at the same time can access a shared database using an appropriate query protocol. Similarly, different levels of architectural description should be allowed to use different architectural idioms. For example, different layers in a layered architecture might be implemented in using different architectural organizations.

The second aspect of heterogeneity is the desirability of combining components that are written in different languages. Since an architectural description is at a higher level of abstraction than the description of the algorithms and data structures that are used to implement the computations, there is no logical reason about why these lower level descriptions must use the same notation. Indeed, module connection systems that support interaction between distinct address spaces often provide this capability (e.g., Unix shell scripts, multiprocess message-passing systems, etc.).

3.6 Analysis

It should be possible to perform rich and varied analyses of architectural descriptions.

Requirements of analysis address the ability to support automated and non-automated reasoning about architectural descriptions. Different architectures permit different kinds of analysis, and it should be possible to tailor the kind of analysis to the kind of architecture. This goes beyond the current support for analysis, which primarily consists of type checking.

When a designer uses a certain set of architectural elements to construct a system it is often because this choice enables analysis of specialized properties of that system. For example, in a pipe and filter architecture, it is possible to analyze properties of throughput, investigate questions of deadlock and resource usage, or infer the input-output behavior of a system from that of the component filters. It should be possible to tailor special-purpose analysis tools and proof techniques to these architectures.

Existing module connection languages provide only weak support for analysis. At best they provide some form of type checking across component boundaries. They rarely permit more semantically-based properties to be analyzed or even expressed. It is possible to add specification of input-output behavior and reason via procedure call proof rules. However, for many other forms of interaction, such as event broadcast, there are currently no corresponding systems of specification and analysis.

The need for enhanced forms of analysis are particularly important for architectural formalisms, since many of the interesting architectural properties are dynamic ones. For example, if a connector is associated with a particular protocol, it should be possible to reason about whether the use of that connector is correct in its context of use. Similarly, issues such as timing, performance, and resource usage may play a significant part in reasoning about whether a given architectural description is adequate.

The variability of kinds of analyses that one might want to perform on an architectural description argue strongly that no single semantic framework will suffice. Instead, it must be possible to associate specifications with architectures as they become relevant to particular components, connectors, and patterns.

4 Problems with Existing Languages

Most existing notations for describing software architectures can be grouped into five broad categories: informal diagrams, modularization facilities provided by programming languages, module interconnection languages, support for alternative kinds of interaction, and specialized notations for certain architectural styles. We now consider some of the ways in which each of these categories fails to satisfy the properties outlined earlier.

4.1 Informal Diagrams

Informal diagrams are typically used to convey a high-level view of system organization. These are frequently drawn as a graph of boxes and lines, where the boxes represent components of many different kinds, and the lines represent interactions. The meanings of both boxes and lines vary considerably from diagram to diagram, and may even vary within one diagram. Different kinds of components in a single diagram are often distinguished with boxes of different shapes, but different kinds of connectives are rarely distinguished visually. For example, lines might represent data flow, control flow, an inheritance relationship, a "contains" relationship, an type-instance relationship, function call, asynchronous message passing, etc.

The problems with such diagrams are relatively obvious. While they may offer a high level of abstraction, their informality is a serious drawback. Although they can convey intuitions effectively, it may be impossible to use them for analysis, the relationship to implementations is tenuous at best, and they are typically constructed form scratch with each new application.

4.2 Modularization Provided by Programming Languages

The second approach to architectural description is to use modularization facilities provided by a programming language. Representative examples include Simula classes, CLU clusters, Alphard forms, and Ada packages. These languages are based on the notion that a module defines an *interface*, which declares (a) the facilities that it is providing to the system (its *exports*) and (b) the external facilities on which it depends (its *imports*). In this context "facilities" refer to the low-level entities (variables, functions, types, etc.) directly supported by the programming language used to define module implementations.

Modularization constructs were originally introduced to partition the code of a system so as to reduce complexity through abstraction, permit cooperative work, and allow incremental compilation of parts of a system. While programming language modules have had some success in raising software system construction to the level of "programming-in-the-large", from the point of view of architectural description they have some serious flaws.

4.2.1 Composition

Most modularization facilities permit hierarchical decomposition: modules may be declared within other modules. However, programming language modules provide poor support for independent composition of architectural elements, and they interfere with separation of structural concerns from programming concerns.

The first problem is that inter-module connection is determined by name matching. It is common for a system to require that elements exported from one module be referred to by the same names in other modules. This is true, for example, for Ada and C. In Ada, modules (packages) are imported in their entirety via a "with" clause. Suppose package Foo declares functions f and g. Then package Baz would use f and g by including the line

```
with Foo
```

at the beginning of the Baz package definition and referring to f and g, with name qualification, as Foo.f and Foo.g, respectively. In C, Foo would be defined by files Foo.c and Foo.h. Then Baz would include the line

```
#include Foo.h
```

in Baz.c (or possibly Baz.h) and refer to functions f and g without name qualification.

This scheme is good enough for the compiler to ensure that addresses match at runtime, but it provides poor support for architectural description. In particular, name matching interferes with independent development of modules by requiring the importing module to use the name by which an element is exported, rather than by a name more appropriately determined by the context of its use.

The second problem is that the use of imports and exports in module interfaces forces system interconnection structure to be embedded in module definitions. Consequently, modules cannot be easily separated from the system in which they were originally defined.

Third, the use of imports and exports confuses algorithmic with architectural description. When facilities are imported from another module, this may indicate interaction between two components of a system. But it might also simply represent the inclusion of lower-level facilities to aid in the implementation of the importing module—for example, by importing a library module.

4.2.2 Abstraction

Programming language modules represent module interfaces as collection of independent procedures, data, and possibly other nameable entities of the languages such as types and constants. As a result, system structure must be expressed in terms of the primitive constructs of the programming language at hand.

Typically, such modules provide only one or two forms of built-in interconnection mechanism. Usually the mechanism is procedure call and data sharing, but it may instead be pipes, message passing, or some other mechanism. While the use of a small set of mechanisms has the advantage of uniformity and simplicity, it also has the significant disadvantage that it preempts any other form of inter-module interaction.

Only rarely do existing systems support a richer vocabulary of architectural interconnection directly. The exceptions are usually in the form of support for one or a few additional simple connections, such as the Ada rendezvous. While these enlarge the vocabulary for architectural description, they do not provide a more general facility for user-definable interactions.

The problem is perhaps not particularly severe at the programming language level, where a simple, uniform computational model is generally to be preferred over a large collection of mechanisms. But the architectural level of design requires a diverse collection of abstractions to represent even the most common forms of component interconnection [Sha93]. Indeed, the interactions represented by lines in informal diagrams are drawn from a much richer and more abstract vocabulary than the language-supported mechanisms. While it is usually the case that most abstractions can be *implemented* by a mechanism such as procedure call or message passing, the inability to use more than the primitive programming-level forms of interaction in system definitions has three negative consequences. First, it tends to force system designers to think of an architecture solely in terms of those primitive constructs. Second, it limits reusability, since components that make different assumptions about module interconnection cannot be included. Third, it limits the level of abstraction that can be used to describe interactions.

The consequence is that a major part of the system design—the description of the high-level interactions between modules—is not explicit. It is encoded in individual procedures calls, and shared data accesses; it is distributed through the code of multiple modules; it often remains undocumented; and it is exceedingly difficult to change.

4.2.3 Reuse

Programming modules provide poor support for reuse for many of the same reasons. Modules drawn from libraries include an explicit import statement (or set of import statements) rather than a requirement that certain specified other capability be provided. This interferes, first, with the implementor's ability to select alternative algorithms and representations and, second, with the possibility of packaging the required capability in different ways.

Further, even at their best, traditional schemes of module description support only reuse of the modules themselves. In particular, there is generally no support for reuse of patterns of composition.

4.2.4 Configuration

As we have indicated, the use of imports and exports leads to a situation in which the connectivity structure of the system is distributed through the module definitions. This makes it essentially impossible for a developer or maintainer to understand or analyze the structure as a whole.

The problem is mildly alleviated if build files, such as those used by Make [Fel79], are used to show dependencies. However, these files are notoriously hard to read and write, they can easily diverge from the actual system, they may show indirect as well as direct dependencies, and they show only the *fact* of a dependency, not the nature of the connection or the designer's intention.

Finally, most module interconnection schemes allow only static configurations: dynamic reconfiguration is not generally supported.

4.2.5 Heterogeneity

Heterogeneity is poorly supported. In general, modules written in different programming languages cannot be combined or can only be combined with special tools for inter-language procedure calls. Further, since only a few primitive forms of module interaction are supported, it is impossible to introduce new kinds of interaction without first encoding them in the primitives at hand. Even worse, conventional programming languages provide no means for distinguishing among different kinds of elements. Since they also fail to support abstractions for interactions, they have no way to express architectural paradigms, let alone ways to combine such paradigms.

4.2.6 Analysis

Programming language modules provide poor support for architectural analysis. This follows in part from the fact, mentioned above, that it is not easy to determine from imports and exports alone what the architectural structure of a system really is. Further, the use of name matching makes it difficult to check for consistency of interconnection. An inter-module connection system should help to ensure that the use of the imported element is consistent with the intentions of its exporter. Attempting to achieve this through name matching is insufficient, since name matching does not ensure proper use. We should require at least correspondence between signatures of functions, structures of types, and so on. Even better, it would be desirable to require consistency between formal specifications.

4.3 Module Interconnection Languages

An alternative to the use of module interfaces for describing system composition is to use a special language for the task. These are sometimes called "module interconnection languages" (MILs) Representative examples of early language designs include MIL75 and Intercol [DK76, Tic79, PDN86]. A more recent example is the module description features of Standard ML (SML) [MTH90].

Module interconnection languages partially separate the description of a system configuration from the parts of the system that are being composed. Some also provide parameterization features for describing templates of system composition, by indicating how a set of module types can be combined to produce new module types. This has been shown to be effective in particular for defining layered systems, such as communication protocol stacks [HL94].

In these respects MILs satisfy some of the properties of composition and configuration better than programming language modules alone. However, they, too, are primarily concerned with resolving name bindings between definitions and uses of low-level programming entities. In particular, they do not change the fact that only low-level interactions (such as procedure call) are supported. They also do not provide a fully general way to indicate reusable patterns of composition. So they share many of the shortcomings discussed in the previous section.

4.4 Languages that Support Alternative Forms of Interaction

Certain notations have been designed to make it easier to describe interactions beyond procedure call and data sharing. For example, Unix provides a shell language that supports direct definition of pipes as connectors [Bac86]. As another example, some systems support event broadcast by extending the facilities of a programming language [SN92, GS93a]. Ada supports intertask communication through rendezvous [DoD83].

Unfortunately, while all of these make it easier to describe *some* high-level interactions, none provide a more general facility for describing new abstractions for interaction. Hence, system builders must encode their intentions in terms of the specific primitives at hand.

4.5 Notations for Specialized Architectural Styles

Recently, a number of systems have been developed to support specific abstract paradigms. Some of these were mentioned in the introduction [Pur88, Kra90, Pou89, Mak92, Bea92, MMHG92]. Each of these works through a specific architectural style in detail. Most provide good high-level support for the paradigm of interest. However, these languages typically assume a homogeneous universe, and live as isolated systems. This makes it difficult to describe the preponderance of real systems that exhibit heterogeneity internally, or that need to interoperate with other systems built in different styles.

5 Conclusions

This paper has examined the need for precise descriptions of software architectures and argued that this need can be addressed as a language design problem. We outlined six critical kinds of properties for architectural description and argued that software architectures require languages that are distinct from the programming languages we have at our disposal. The core of the argument is that the semantic entities for architectural description are different and, further, we need a higher level of abstraction than conventional programming languages can provide.

In recognition of this fact, several new languages have recently been proposed to address the needs of architectural description. Rapide [LAK$^+$95] provides a language whose interface model is based on SML augmented with events and event patterns. Event patterns can express richer aspects of module interaction than procedure call, and can also be used to define system communication topologies. UniCon [SDK$^+$95] provides an architectural description language in which both components and connectors are defined as first class compositional entities. Wright [AG94a, AG94b] is an architectural specification language that makes the notion of first-class connection precise by defining the semantics of connectors as formal protocols in a variant of CSP [Hoa85]. The Aesop System [GAO94] provides a general framework for defining many architectural languages, each specialized to a particular architectural style. The core of Aesop is a generic architectural description language called Acme, from which the other more specialized forms are developed. All of these language proposals are currently at the level of research prototype: it remains to be seen how well they will address the requirements for architectural description as they become more mature.

Acknowledgements

We would like to thank the numerous readers of earlier drafts of this paper for their helpful comments. In particular, we thank Nancy Mead, Robert Schwanke, Dilip Soni, Will Tracz, and the participants of the CMU Software Architecture Reading Group.

References

[AG94a] Robert Allen and David Garlan. Beyond definition/use: Architectural interconnection. In *Proceedings of the ACM Interface Definition Language Workshop*, volume 29(8). SIGPLAN Notices, August 1994.

[AG94b] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*, chapter 5.12, pages 111–119. Software Series. Prentice-Hall, 1986.

[Bea92] Brian W. Beach. Connecting software components with declarative glue. In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

[D+91] Doubleday et al. Building distributed Ada applications from specifications and functional components. In *Proceedings of TRI-Ada'91*, pages 143–154, San Jose, CA, October 1991. ACM Press.

[DK76] 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.

[DoD83] Reference manual for the Ada programming language, January 1983. United States Department of Defense.

[Fel79] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software—Practice and Experience*, 9:255–265, Nov 1979.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*. ACM Press, December 1994.

[GD90] David Garlan and Norman Delisle. Formal specifications as reusable frameworks. In *VDM'90: VDM and Z – Formal Methods in Software Development*, Kiel, Germany, 1990. Springer-Verlag, LNCS 428.

[GKN88] David Garlan, Gail E. Kaiser, and David Notkin. On the criteria to be used in composing tools into systems. Technical Report 88-08-09, Department of Computer Science, University of Washington, August 1988.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, pages 31–44. Springer-Verlag, LNCS 551, October 1991.

[GS93a] David Garlan and Curtis Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD, May 1993.

[GS93b] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering, Volume I*, New Jersey, 1993. World Scientific Publishing Company.

[GSO+92] David Garlan, Mary Shaw, Chris Okasaki, Curtis Scott, and Roy Swonger. Experience with a course on architectures for software systems. In *Proceedings of the Sixth SEI Conference on Software Engineering Education*. Springer Verlag, LNCS 376, October 1992.

[HL94] Robert Harper and Peter Lee. Advanced languages for systems software : the fox project in 1994. Technical Report CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Kra90] Jeff Kramer. Configuration programming – a framework for the development of distributable systems. In *Proceedings of the IEEE International Conference on Computer Systems and Software Engineering*, Israel, May 1990. IEEE.

[L+88] K.J. Lee et al. An ood pardigm for flight simulators, 2nd edition. Technical Report CMU/SEI-88-TR-30, Carnegie Mellon University, Software Engineering Institute, September 1988.

[LAK+95] David C Luckham, Lary M. Augustin, John J. Kenney, James Veera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering, to appear*, 1995.

[Mak92] Victor W. Mak. Connection: An inter-component communication paradigm for configurable distributed systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, UK, March 1992.

[MMHG92] LTCC Erik Mettala and eds. Marc H. Graham. The domain-specific software architecture program. Technical Report CMU/SEI-92-SR-9, Carnegie Mellon Software Engineering Institute, June 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

[PDN86] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[Pou89] D. Pountain. Occam II. *Byte*, 14(10):279–284, October 1989.

[Pur88] James M. Purtilo. A software interconnection technology. Technical Report UMIACS-TR-88-83 CS-TR-2139, University of Maryland, College Park, November 1988.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[Ros85] Frederick Rosene. A software development environment called STEP. In *Proceedings of the ACM Conference on Software Tools*, April 1985.

[SDK+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering, to appear*, 1995.

[Sha89] Mary Shaw. Larger scale systems require higher level abstractions. *Proceedings Fifth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes*, 14(3):143–146, May 1989.

[Sha91] Mary Shaw. Heterogeneous design idioms for software architecture. In *Proceedings of the Sixth International Workshop on Software Specification and Design, IEEE Computer Society, Software Engineering Notes*, pages 158–165, Como, Italy, October 25-26 1991.

[Sha93] Mary Shaw. Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status. In *Proceedings of the Workshop on Studies of Software Design*, May 1993.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[Tic79] Walter F. Tichy. Software development control based on module interconnection. In *Proceedings of the Third International Conference on Software Engineering*, pages 29–41. IEEE Computer Society Press, May 1979.