Three Concepts of System Architecture

David C. Luckham James Vera Sigurd Meldal *

July 19, 1995

Abstract

Abstract

An *architecture* is a specification of the *components* of a system and the *communication* between them. Systems are constrained to conform to an architecture. An architecture should guarantee certain behavioral properties of a conforming system, i.e., one whose components are configured according to the architecture. An architecture should also be useful in various ways during the process of building a system.

This paper presents three alternative concepts of architecture: *object connection architecture*, *interface connection architecture,* and *plug and socket architecture.* We describe different concepts of *interface* and *connection* that are needed for each of the three kinds of architecture, and different conformance requirements of each kind. Simple examples are used to compare the usefulness of each kind of architecture in guaranteeing properties of conforming systems, and in correctly modifying a conforming system.

In comparing the three architecture concepts the principle of *communication integrity* becomes central, and two new architecture concepts, *duality* of sub-interfaces (*services*) and connections of dual services (*service connection*), are introduced to define plug and socket architecture. We describe how these concepts reduce the complexity of architecture definitions, and can in many cases help guarantee that the components of a conforming system communicate correctly. The paper is presented independently of any particular formalism, since the concepts can be represented in widely differing architecture definition formalisms, varying from graphical languages to event-based simulation languages.

1 Introduction

In recent years object-oriented techniques have caused a paradigm shift in systems development. As a result, we have become much more efficient at building objects — the components of systems. So much so, in fact, that in just ten years since Ada83 [15] was first adopted as the DoD standard programming language, it was deemed necessary to undertake a large redesign (Ada9X [7]) to incorporate OO technology.

However, while object-oriented techniques have made us better at building components, they do not solve the conspicuous problem of combining components into complete systems. Indeed large scale software systems continue to be plagued with a variety of development problems which have been amply discussed in the literature (see, e.g., [5]).

There is now a widespread belief that software engineering must go beyond object oriented methods to a new technology based upon "architecture". Unfortunately, it is fair to say that the concept of architecture, although widely used, has no generally accepted precise definition. Often the term has little meaning other than a graphical diagram consisting of boxes and arrows that serves as a pictorial table of contents for detailed design and code documents.

The most common answer given to the question "*what is architecture*?" is "*interfaces* and *connections*". This is not a bad answer, but the devil is in the details of what the properties of "interfaces" and "connections" are, how they are related to the system, and what use can be made of them.

*This project is funded by DARPA under ONR contract N00014-92-J-1928 and AFOSR under Grant AFOSR91-0354

The general expectation is that an *architecture* specifies the modules (or objects) of a system (called the *components* of the system), and how the components interact so as to satisfy the requirements of the system. However, most people go beyond this definitional goal to postulate that architectures are also useful in building or managing a system.

So there is also an expectation that an architecture should be a template for guiding the development of a system (or family of systems) to satisfy a given set of behavioral requirements. As a template, an architecture defines constraints on a system. A system "has" that architecture if it conforms to those constraints (in a sense to be defined), and if it does, then it meets the given requirements. How an architecture can be used to develop systems depends upon *(i)* the kind of information it provides about a system, *(ii)* a precise definition of the conformance relationship between architecture and system, and *(iii)* the kinds of automated tools that can be constructed to support the use of architecture. For example, an architecture could contain enough information to allow simulation to assess system behavior and performance before the system is built. It could define behavioral constraints that allow formal reasoning to establish that a system conforming to that architecture will satisfy the system's requirements. Or it might contain information that allows changes to a system to be made, so as to maintain required behavior, on the basis of analyzing the architecture alone.

Certainly, as a minimal expectation, an architecture should guarantee important properties of a system that is built to conform to it. Conformance should be testable at every stage in system development.

Given the range of possible roles of architecture in building systems, the primary issue is to clarify the concept itself. We should be able to give some answers to the following questions:

- does every system have an architecture?
- what is an architecture — how does it differ from a system?
- how does an architecture constrain the system?
- how can an architecture be used to build or modify systems?
- how should architecture definition languages (abbrev. ADLs) differ from programming languages?

In this paper an architecture is assumed to be a specification of a class of systems at some level of abstraction. An architecture consists of *interfaces*, *connections*, and *constraints*. The interfaces specify the components of the system, and the connections and constraints define how the components may interact. The main relationship between an architecture and a system is *conformance*. Conformance defines when a system has (or conforms to) an architecture. For our discussion it is not important whether an architecture is an actual part of a system, as it is in some views, or whether it is totally separate from a system, such as a diagram on paper.

We present three concepts of *architecture*: *(i) object connection architecture*, *(ii) interface connection architecture*, and *(iii) plug and socket architecture*. The first kind of architecture is typical of object-oriented systems in the current O-O programming languages. It is probably what most people who answer "yes" to the first question above have in mind. The second kind is typical of systems (whether object-oriented or not) that are developed from architectural plans, such as communication protocols and hardware. The third kind of architecture is typical of hardware, but we shall argue it should be used to plan many software systems. The first two concepts of architecture differ in how they constrain a system that conforms to that architecture, how they facilitate the construction of a system, and the properties they can guarantee that conforming systems will possess. Plug and socket architectures are a specialization of interface connection architectures to deal with issues of *(i)* scalability (i.e., defining large architectures), and *(ii)* guaranteeing properties of conforming systems by simple tests on the architecture only.

Our three definitions of architecture require progressively more sophisticated interfaces and connections. We show how the object-oriented concept of interface is used in architectures of the first kind, but needs to be extended with new kinds of features for the second kind, and organized into groups of features ("plugs" and "sockets") for the third kind. We describe how the concept of connection also changes. The new kinds of interfaces and connections lead us to suggest, in general terms, some new features for ADLs.

These three concepts of architecture are not restricted in any sense to being static. The number of interfaces is not bounded a priori, and connection definitions are not restricted to defining interactions between finite sets of components.

We define three basic conformance criteria :

1. **decomposition** : for each interface in the architecture there should be a unique module corresponding to it in the system (i.e., the component implementing that interface). [1]
2. **interface conformance**: each component in the system must conform to its interface. We assume that behavioral constraints can be part of interfaces, so this constraint is, in general, stronger than the syntactic interface conformance usually required by programming languages.
3. **communication integrity**: the system's components interact only as specified by the architecture.

We use two common system maintenance tasks to illustrate how interface connection architectures are more useful than object connection architectures for manipulating conforming systems. This discussion leads us to formulate properties that interface connection architectures should possess in order to guarantee that systems conforming to them will also possess required properties: *ideal interfaces*, *connectability* of interfaces, and *correct connections*.

However, determining if an architecture possesses these properties, and also determining if an arbitrary system conforms to an architecture, are often difficult problems, and in the general case, undecidable. Moreover, we discuss how interface communication architectures can inherit the complexity of the communication structure in a system. These considerations motivate a search for new concepts of architecture in which some of these problems are easier to solve.

Plug and socket architectures are a specialization of interface connection architectures to deal with issues of *(i)* scalability (i.e., defining large architectures), *(ii)* refinement (i.e. the process of defining an architecture), and *(iii)* guaranteeing properties of conforming systems by simple tests on the architecture (rather than the systems as implemented). We show how the interface concept as used in interface connection architectures should be extended with a new ADL concept of *duality* of features. This allows interfaces to be organized into "dual" groups of features ("plugs" and "sockets") to enable the definition of plug and socket architectures.

Finally we note that recent work has attempted to define popular categories of architectures (e.g., client-server, pipe-and-filter, etc., see [4]). These categories depend upon the roles and properties of various components of the architecture. Here we are concerned with defining concepts of architecture that apply to any category.

2 Some Terminology

To discuss concepts of architecture we need a small amount of terminology. The following concepts are explained in general terms so as not to imply any particular language or semantics; different languages, (e.g., C++ [3], VHDL[6], Ada [15], Rapide [10], [9]) define these concepts using their own semantics. Also, to keep our discussion independent of any language, examples are given using a syntax that is easily understood by anyone familiar with C++, VHDL, or Ada.

feature	= a computational element of a module,	e.g., a function or port or action.
uses	= makes use of a feature,	e.g, function call, send message to a port.
specification	= a set of constraints expressed in a formal language.	
interface	= a definition of a set of features, together with a specification of their behavior.	e.g., public part of a C++ class, or an Ada package specification
module	= a computational entity containing the set of features defined in an interface; modules may compute independently of, or concurrently with, one another.	e.g., an Ada package body.
conformance	= a module *conforms* to an interface if it contains the features specified by the	

[1]More sophisticated decomposition criteria, such as requiring an abstraction mapping from sets of system components to interfaces in the architecture, are beyond the scope of this presentation.

	interface so that they have the behavior specified by the interface; this is called *interface conformance*.	
object connection	= a relation between modules implying interaction,	e.g., one module uses a feature of another module.
architecture	= a definition of a set of modules and the interactions between them, by means of interfaces, connections, and constraints.	
component	= a module corresponding to an interface in an architecture.	

3 Object Connection Architectures

An *object-oriented interface* specifies the features that must be *provided* by modules conforming to the interface. Typically, a feature is a function, an interface specifies the name and signature of the function, and a module contains an implementation of a function with that name and signature. Examples are C++ classes, Ada package specifications, and VHDL entity interfaces.[2]

An *object-oriented system* consists of *(i)* object-oriented interfaces, *(ii)* a set of modules, each conforming to an interface by providing an implementation of the features specified in that interface, and *(iii)* connections defined by the uses each module makes of the features specified in the interfaces of other modules. An *object connection architecture* consists of the interfaces and connections of a system. An object connection architecture is depicted in Figure 1. In this figure, interfaces are shaded "lids" of the boxes, and the boxes represent modules that conform to the "lids"; connections are shown as directed arcs from a module to a feature in an interface of another module. The direction of a connection indicates which interface specifies a feature being used; the dotted arrows indicate the association between an interface feature and its implementation in a module (often simply association by name).

Figure 1 could represent a C++ program, for example, in which features are functions, interfaces are classes (public parts), and connections are function calls.

The architecture in Figure 1 is called an *object connection architecture* because the connections are from object to object. It is, in fact, part of the system. It has the major disadvantage that the modules must be built before the architecture is defined, so it cannot be used to plan the system. It is, however, a very common (mis)conception of "architecture".

Conformance to an object connection architecture is usually enforced by the language in which the system is programmed. For example, decomposition is satisfied in most programming languages by rules for associating an interfaces with a unique module; similarly, communication integrity is enforced by the rules of visibility between modules. Interface conformance is the only non-trivial constraint which is usually only partially checked by compiletime checking of syntactic rules; conformance to semantic constraints is not guaranteed.

Example

To illustrate how the ease or difficulty of determining properties of a system depend upon the kind of architecture the system has, we take the well-known paradigm example of a compiler. Our simple compiler has the architecture shown in Figure 1. It consists of three modules, a Parser, a Semanticizer and a Code Generator. Its interfaces are shown in Figure 2. The interfaces consist of only a subset of the features a "real" compiler implementation might have, but suffice to illustrate how the kind of architecture impacts on commonly practiced modifications to the system.

What kinds of uses might we want to make of our architecture?

Problem 1: Determine which modules are affected by a change to an interface. For example we might wish to remove the function Incremental_Semantize from the Semanticizer interface. To determine the effects of this change we must determine which modules are connected to

[2]These examples of interfaces differ in some respects, e.g., packages and entities are not types in Ada or VHDL; but each allows the interface features of a module to be visible outside the module, and permits the module to encapsulate or hide other features from outside use.

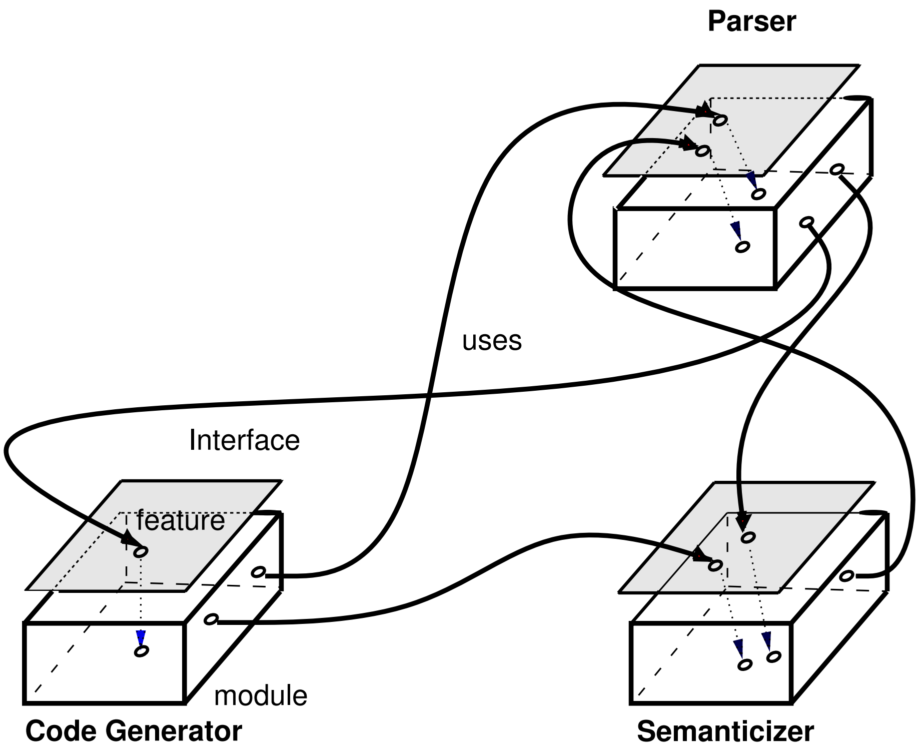

Figure 1: An object connection architecture.

Incremental_Semantize. To do this we need to examine each of the modules to see if any of them uses this function. So, we must examine all modules in the system.[3]

Problem 2: Determine whether we can replace a module by another one that conforms to the same interface.

Imagine then that we want to change our compiler by substituting a new module for the Code_Generator interface instance G. It conforms to the Code_Generator interface just as the module it replaced did. Object-oriented subtyping such as in C++ allows this kind of change. But does the changed system have the same functional behavior? From Figure 1 we see that the original Code_Generator module made a call to the Initialize function of the Parser. The new Code_Generator module is not required by its interface to make this call and thus might not. The failure to call the Initialize function would likely result in the misbehavior of the Parser and the failure of the entire system. So even interchanging modules that conform to the same interface may have disastrous effects.

□

[3]This issue is distinct from that addressed by the *with* clauses of Ada programs. The Ada *with* clause indicates possible compilation unit dependencies for separate compilation.

```
class Parser is
  function Initialize();
  function FileName() return String;
end Parser;

class Semanticizer is
  function Semantize(Tree);
  function Incremental_Semantize(Context :Tree, Addition : Tree);
end Semanticizer;

class Code_Generator is
  function Generate(Tree);
end Code_Generator;

P: Parser; S: Semanticizer; G: Code_Generator;
```

Figure 2: Compiler module interfaces and instances.

This example illustrates that we cannot rely only upon the object connection architecture of a system to determine the effects on the system of changing an interface or interchanging modules that conform to an interface — two relatively simple and desirable uses of architecture.

4 Interfaces for Architecture

Ideally, an interface should ensure substitutivity of modules that conform to it.
The ideal interface principle:

$I(x) \equiv I(y) \Leftrightarrow \forall P(x), P|_y^x \sim P$

which we read as "If the interface of module x is the same as the interface of module y, then for any program P which contains module x and can be shown to behave in some required manner on the basis of the information in the interface alone, then module y could be substituted for x and the resulting program would satisfy the same behavioral requirements."[4]

This principle could be generalized further to encompass subtyping:
The ideal subtyping principle:

$I(y) <: I(x) \Rightarrow \forall P(x), P|_y^x \sim P$

As the previous example shows, the use of an interface feature, such as whether it is called, can be crucial in interchanging modules for the same interface. To specify such use, the interface must contain specifications of the semantics of its features. Thus, the ideal interface principle, requires that an interface must contain specifications of the semantics of its features and not just signature information. Unfortunately, it is an undecidable problem whether a module conforms to a semantic specification given a sufficiently powerful specification language. But even if conformance with specifications of provided features was decidable, object-oriented interfaces cannot satisfy the ideal principles.

Part of the failing of object-oriented interfaces is that they specify only the features *provided* by modules. Connections cannot be defined in terms of interfaces. This forces architecture connections to be object to object. Consequently, when objects implementing the same interface are interchanged, the connections may change. Therefore, the concept of interface in ADLs should define also those features that modules *require* from their environment. ADL interfaces should specify five characteristics:

1. A list of the features provided by the module,
2. A list of the features required by the module,
3. A specification of the behavior of the provided features,

[4] I. e., the behavior of the new program satisfies the requirements that were satisfied by the previous program.

4. A specification of the behavior of the required features.
5. A specification of the mutual interaction between two or more features (e.g., the use of one feature changes the behavior of other features).

By specifying all of the required features used by a module, an ADL interface explicitly expresses the dependencies a conforming module has on its environment. We say the interface is *contextualized*. Specification of interactions between features adds additional constraints on how a module depends upon context; e.g., the Parser interface in Figure 2 could specify that Initialize must be called before calls to FileName will return values.

Object-oriented interfaces such as in C++ have traditionally only contained characteristic 1, with characteristics 2, 3, 4 and 5 left to informal comments or buried in the implementation of the interface. As a result, the object-oriented notion of subtyping (substitutability) is based only on characteristic 1. This is probably due in part to the influence of the original Simula Class design [11], but also due to an emphasis on the design of the interfaces to support *information hiding* [12]. We argue that in leaving out the features a module requires, object-oriented interfaces have hidden too much to be useful for architecture definition, as illustrated by the previous example.

Interfaces that specified both provided and required features were part of the Euclid language design in the 1970's [8]. The motivation of Euclid was to support abstract data types and program verification [2]. Our motivation here is to support definition of architectures that can be used to plan and guide system development.

5 Interface Connection Architectures

An interface connection architecture defines all connections between the modules of a system using only the interfaces. Interfaces specify both *provided* and *required* features. Such an architecture, and a system conforming to it, is shown in Figure 3.

Connections are defined between a required feature and a provided feature. A connection from a using module to an interface feature of a providing module in Figure 1 is represented in Figure 3 as an arc from a required feature specified in the interface of a user to a provided feature specified in the interface of a provider. Thus all of the connections between objects in the object connection architecture are defined as *connections between interface features* in the interface connection architecture. This is possible because ADL interfaces specify *required* features.

Two new ADL mechanisms are needed to define connections in interface connection architectures that are not available in present-day programming languages such as C++ or Ada :[5]

- A mechanism to define connections between required features of one interface with provided features another interface. Connections could be as trivial as identifying pairs of names in interfaces (having a semantics similar to pin connections in hardware modeling languages), or as powerful as the pattern triggered reactive rules of Rapide [10]. In examples we assume that some sort of feature-to-feature identification is used, shown in Figure 4 as "connect ... ".
- A mechanism to allow a module to "use" the required features of its own interface. This is shown in Figure 3 by dashed arrows between the modules and their interfaces. An example of this mechanism could be function call, but now the only way a module can invoke non-local functions is through its own interface. The architecture must then connect the module's interface functions with interface functions of other modules.

The semantics of *connection* changes from that used in object connection architectures:

interface connection = an association between a required interface feature, $\overline{f}$, and a provided interface feature, f, such that any use of $\overline{f}$ is replaced by a use of f. E.g., calls to function $\overline{f}$ become calls to f.

[5]Simulation languages such as Verilog and VHDL do contain rudimentary forms of such features.

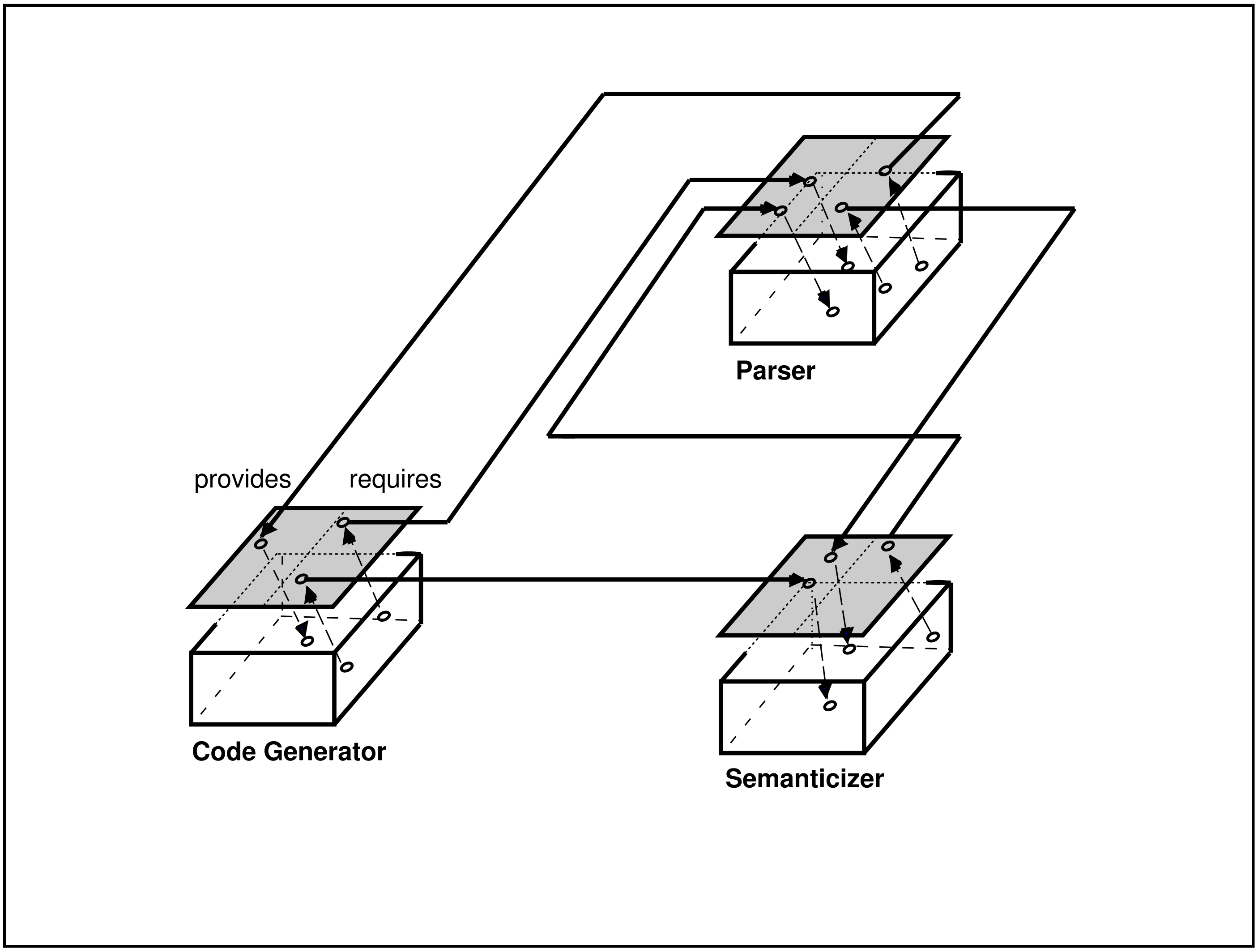

Figure 3: An interface connection architecture and system.

An interface connection architecture constrains the modules of a system to satisfy the three conformance criteria (Section 1). In particular, ensuring communication integrity is no longer trivial. If communication integrity is not ensured, the ability to manipulate the system using only information in its architecture, as illustrated below, will be lost.

Examples

Our compiler example might be rewritten with contextualized modules as shown in Figure 4. We write "provides" and "requires" to separate the provided and required features. We also use a simple syntax to connect required features to provided features.

The interface connection architecture of our new compiler system is depicted in Figure 3. The modules no longer use features of other modules directly. Instead the interfaces specify exactly the features provided and required by each module. The compiler should function exactly as before, but now the connections between the modules are expressed in the interface connection architecture *before* the modules are built. Also, since required features are specified, formal specifications in interfaces can

```
class Parser is
provides:
   function Initialize();
   function FileName() return String;
requires
   function Semantize(Tree);
   function Generate(Tree);
specification ...
end Parser;

class Semanticizer is
provides:
   function Semantize(Tree);
   function Incremental_Semantize(Context : Tree; Addition : Tree);
requires:
   function FileName() return String;
specification ...
end Semanticizer;

class Code_Generator is
provides:
   function Generate(Tree);
requires:
   function Initialize_Parser();
   function Semantize(Context : Tree; Addition : Tree);
specification ...
end Code_Generator;

P: Parser; S: Semanticizer; G: Code_Generator

Connect
  P.Semantize to S.Semantize;
  P.Generate to G.Generate;
  S.FileName to P.FileName;
  G.Initialize_Parser to P.Initialize;
  G.Semantize to S.Incremental_Semantize;
```

Figure 4: An interface connection architecture for a compiler

specify relationships between the provided and required features as well as properties of each feature individually. This allows more powerful specifications in interfaces than before, so we have indicated interface specifications in Figure 4 by including a "specification" part of the interface definition.

The interface connection architecture can be used to analyze the two problems we raised previously. How difficult are our two problems now?

Problem 1: We can determine potential users of a provided feature in an interface simply by inspecting the architecture connections. Interfaces whose required features are connected to the provided feature are interfaces of modules that are potential users. So connections are used to bound the search. For example to determine the users of function Semantize provided by the Semanticizer S, we check if there is a connection from a required function of either the Parser or Code_Generator. Note that a connection from either of the required functions, Semantize in the Parser P or Code_Generator G, would satisfy the function subtyping constraint (below) that is a necessary condition for correctness of connections – a call to either of these would supply the Tree parameter needed for the successful evalutation of the provided function in the Semanticizer; however, a connection from, say the required function Initialize_Parser in G, could not be correct because that function is not a supertype of the provided function.

To determine if a potential user actually uses an interface feature we would need to analyze specifications in the interface of the user — and assume that modules satisfy the semantic specifications in their interfaces.

In an object connection architecture we have to inspect all modules to find potential users, even if interfaces contained specifications because they could only identify the provided features. Then we have to analyze those modules to see what use they make of the feature.

Problem 2: Assuming that the compiler's modules satisfy the communication integrity constraint, we can interchange any two modules that conform to an interface. This will result in a system that behaves exactly as the previous one whenever that behavior is logically implied by the specifications in the interfaces and the connections. So, we are assuming that the modules satisfy their interface specifications, and, given a particular desired behavior, that the interface specifications together with the connections, imply that behavior.

For example, we want to replace the Code_Generator module as before, and we are concerned about its effect on initializing the Parser. The code generator's interface has a required function, Initialize_Parser, that is the same type as the provided function, Initialize, in the Parser interface. A correct connection between those two functions is consequently possible. If such a connection does not exist, then neither the old nor new Code_Generator module can initialize the Parser (by communication integrity), so the system must function correctly without initializing the Parser. If there is such a connection, then we inspect the specifications in the Code_Generator interface. If they imply that a call must or must not be made, then both the new and the old modules are obliged to satisfy the constraint – and we can be sure that the Parser initialization requirement is not violated when the exchange occurs. If there is no constraint about calling Initialize then we assume that the system will function correctly in either case.

□

This example illustrates a general approach to our two common problems: they can be resolved in systems with interface connection architectures by analysis of only the interfaces and connections. Therefore, we argue, these problems are easier to solve in systems with interface connection architectures than object connection architectures. But our solutions only work if we assume that *(i)* modules conform to interfaces, *(ii)* the system's behavior that we want to preserve is implied by the constraints on the interfaces and connections of the architecture, and *(iii)* modules obey communication integrity. Although these assumptions can be checked in many cases, there is no general algorithm for doing so.

6 Satisfying Communication Integrity

Communication integrity implies that all communication between components of a system is specified by the connections of the interface connection architecture. This is a powerful constraint on the implementation of a system. Note, however, that only communication between *components* is constrained. For example, a component could directly call a data structure say, an arithmetic module, that was not a component (i.e., did not correspond to an interface of the architecture) provided this data structure is not used to communicate with another component.

There are many approaches to ensuring that a system maintains the communication integrity of its interface connection architecture. These can involve language restrictions, or various "sufficient" programming style rules. We discuss some approaches briefly here.

Strict Interface Visibility: *A component of a system is allowed to interact with other components only by using its own interface features.*

This restriction guards against object connections being programmed between components where there is no connection between those components in the architecture. If the system is programmed in

C++, for example, the normal visibility rules of C++ would have to be restricted where they applied to system components. An object that is a component could only access functions of another component by calling functions in its own class interface. C++ classes would need to be extended to declare required functions. [6]

If the features of a module are used by two or more components of a system, or the module (or an access to it) is passed as a parameter of communication between components, then we say that the module is *shared* between the components.

If a module is shared by components it should have a *safe interface*:

safe interface = an interface with constraints requiring that the use of a feature does not influence the behavior of future uses of that or other features of the interface.

A safe module has no state whereby results of using its features can be influenced by prior uses. Examples of safe modules are stateless packages such as an arithmetic module, and modules implemented in the function-oriented programming style.

Safe Sharing: *A module is safely shared between components of a system if either it is a component and those components are all connected to it, or the module is safe.*

This restriction attempts to guard against a module being used to establish communication paths that are not defined in the interface connection architecture. Unspecified communication could take place either by components directly using features of a module, or by passing a module (or pointers or access rights to it) to one another.

A module conforming to a safe interface may be shared between components without being represented in the architecture.

When an interface is *not* safe it is called a *communication interface:*

communication interface = an interface allowing the use of a feature to influence the behavior of future uses of that or other features of the interface.

A communication module is one that has a communication interface, e.g., a file system. If a file system is shared between components, they may communicate by reading from and writing to shared files.

As a consequence of *safe sharing*, a communication module that is shared between components must be a component to which the components that share it are connected.

Communication integrity forces an interface connection architecture to define the communication of a system "completely". If system components are to share a file system, for example, the possible communication between them must be represented. This raises issues of scale and complexity of interface communication architectures. This is good, because now we are forced to face some of the complexity issues of systems at the architecture design stage. Such issues are hidden by object connection architectures, and become sources of error for portability and composition of systems.

7 ADL Features for Scale and Connectability

Interface connection architectures allow us to define the interfaces and connections of a system *before* building the system. The advantage is that they can be used as a plan to study and develop a system. However new issues arise that are not usually recognized as problems for object connection systems, but become obvious in interface connection architectures. These fall into two categories, issues of *scale* and *connectability*:

1. Proliferation of interface features.

[6] For example, required functions could be introduced into C++ by a convention of declaring function pointers in a class public part and objects of the class assuming that functions are assigned to those pointers by other objects.

2. Proliferation of connections.
3. Connectability of interface features.
4. Correctness of connections.

Scale poses a new issue for ADLs: how to express succinctly large numbers of features in interfaces, and connections between them. Architectures at detailed levels of abstraction can often have on the order of 100 features in an interface with each feature being connected to several others in different interfaces (e.g., the Sparc V9 instruction set architecture in Rapide [13]). This situation has been accentuated by introducing *requires* features into interfaces, and also by making the connections between interfaces explicit — whereas they were implicitly defined inside the modules of object connection systems. Proliferation of interface ports and port-to-port connections is already a problem in representing hardware models in present simulation languages such as Verilog [14] and VHDL [6][7].

To put it simply, an object connection system will hide a complex spaghetti of connections inside the objects, whereas an interface connection architecture will raise the level of visibility of connections so that they become a prominent property of the architecture. Indeed, complexity of connections is exactly the kind of issue one wants to reveal as early as possible in system planning. But now, ADLs must find ways of coping with the potentially large number of features of an interface.

A second issue, related to scale, is *connectability* of interface features. An ADL should not allow interface features to be connected if there is no possibility that the resulting communication will make any sense — e.g., if the data communicated would violate type requirements at either end of the connection or if two ends of a connection obey incompatible protocols. Connectability is a concept that depends upon the particular semantics of *interface feature* and *connection*. It should be a necessary condition for correctness of connections, and should be both powerful enough to disallow many kinds of erroneous connections, and easily testable, say by compile time checks. An ADL should have a semantics which is sufficiently powerful to allow these two concepts to be defined:

connectability of interface features	=	an easily checked property of interface features necessary for correct connection, e.g., the types of functions, or the signatures of ports.
correct connection	=	a connection between two interface features such that the implied transfer of data is consistent with all constraints on the features.

Correctness of a connection is a logically stronger concept than *connectability*. Correctness implies that if a connection results in data flow, the data satisfies the data type requirements of features at either end of the connection, and also satisfies any semantic constraints required of the connected features. Correctness will usually be hard to determine in specific architectures, and is an undecidable problem in the general case.

Examples

1. Suppose interface features are functions. Each function has a type determined by the type of its arguments and the type of its return value. Connectability could be defined as the type of a *provides* function g must be a subtype of the type of a *requires* function, f, i.e., $Type(g) <: Type(f)$. Suppose a connection between f and g is defined as identification of f with g, so that any call to f is replaced by a call to g with the same arguments. For example, a required function f with no arguments whose return type is Integer could be connectable to a provided function g of no arguments whose return type is Positive_Integer. The reverse however would not be connectable since any given Integer value may not be acceptable when we require a Postive_integer. Also, a required function with no arguments whose return type is Integer and who's semantics is that it returns only prime numbers, is connectable to a provided funtion with no arguments whoe return type is Integer and who's semantics is that it returns only even numbers; but such a connection would not be correct.

[7] E.g., VHDL has an iterative loop-like *generate* construct to define connections over enumerable sets of entities.

2. Suppose interface features are ports as in VHDL. A port has a mode, *in* or *out*, and a type that defines the data carried on the port. (Here, *out* and *in* correspond to *provides* and *requires*.) Connectability could be defined requiring an *out* port to be connected with an *in* port. A correct connection is one between an *out* port of one entity interface and an *in* port of another entity interface where both ports are of the same type.
3. In a graphical formalism for defining architectures, features could be represented by colored shapes such as circles and triangles in boxes representing interfaces. Two features might be connectable if they have the same shape. Connections could be represented as directed lines joining features in different boxes. A connection might be correct only if the features it connects are the same shape and color.
4. In [1] a feature is a port represented as a CSP process, which may be viewed as a formal specification of the events at the port. A connector is a set of *role* processes together with a process that coordinates (or schedules) the roles, called the *glue*. The concept of connectability is associated with a port and a role: the port process must be a refinement of the role process. A correct connection between a set of ports by means of a connector is a binding of ports to roles of the connector to which they are connectable, so that the instance of the connector has certain properties, freedom from deadlock being one of them.

□

In the first and last examples above we see that, in general, correctness of a connection in an architecture depends upon the semantics of the features it connects, and not simply upon some syntactic property such as the signature of a function. Just as determining whether or not a module in a system conforms to its interface is an undecidable problem, so is determining the correctness of a connection in an architecture, given a reasonably rich language for expressing feature semantics. Thus we must expect connectability to depend only upon syntactic properties in order to be easily checkable. This situation is analagous to type compatibility in languages like C++, which is automatically checked by the compiler, whereas type correctness, which takes account of the semantics of the operations of the types (see Section 4), is usually not checked at all although in many cases it could be. Similarly, although in the general case correctness of connections is an undecidable problem, we believe that for many kinds of real life architectures it can be checked automatically.

8 Duality and Interface Services

In general, communication between modules in a system involves sets of connections between sets of interface features of the modules. Often, there is a protocol relating the order and dataflow between features in the sets. Correct connection must satisfy both the formal constraints on individual interface features and the protocols relating dataflow among the different features in the set.

Here we introduce two new ADL concepts, *duality* and *services*. The purpose of these concepts is to structure interfaces into sets of features (called services). In Section 9 we define a concept of a connection between a pair of dual services which denotes a set of connections between interface features in the two services. Service connections can often be easily checked for correctness.

dual features	=	features in different interfaces that can be correctly connected. E.g., required and provided functions of the same type with the same formal constraints.
service	=	a set of *provides* and *requires* interface features and formal constraints. E.g., a contextualized interface with constraints.
dual services	=	two services each consisting of a set of features each member of which is a dual of a feature in the other service. E.g. hardware plugs and sockets.

Dual features and dual services are easily recognized in many practical cases, e.g., if two features are connectable and have the syntactically identical constraint. If correct connectability is a symmetric relation among features, so also is duality of features and services, and furthermore, a feature or service is a dual of its duals.

We note that a pair of dual services defines a bi-directional communication subsystem containing both the signatures of the features in the subsystem and the semantics of their use.

Next, we allow services to appear in interfaces. That is, the ADL concept of contextualized interface is extended so that services can appear in interfaces (see Figure 5). A service is always a *provided* feature of an interface [8]. The semantics of a service is that it is simply a device for structuring an interface into related groups of features, each group having its own formal constraints. Thus, an interface containing services can be "flattened", i.e., it is equivalent to an interface that consists of the sets of provides and requires features of each of its services together with their constraints, each feature being suitably named to prevent name clashes with features from other services. The concept of services may be realized in a particular ADL in different ways, e.g., services might simply be interfaces within interfaces.

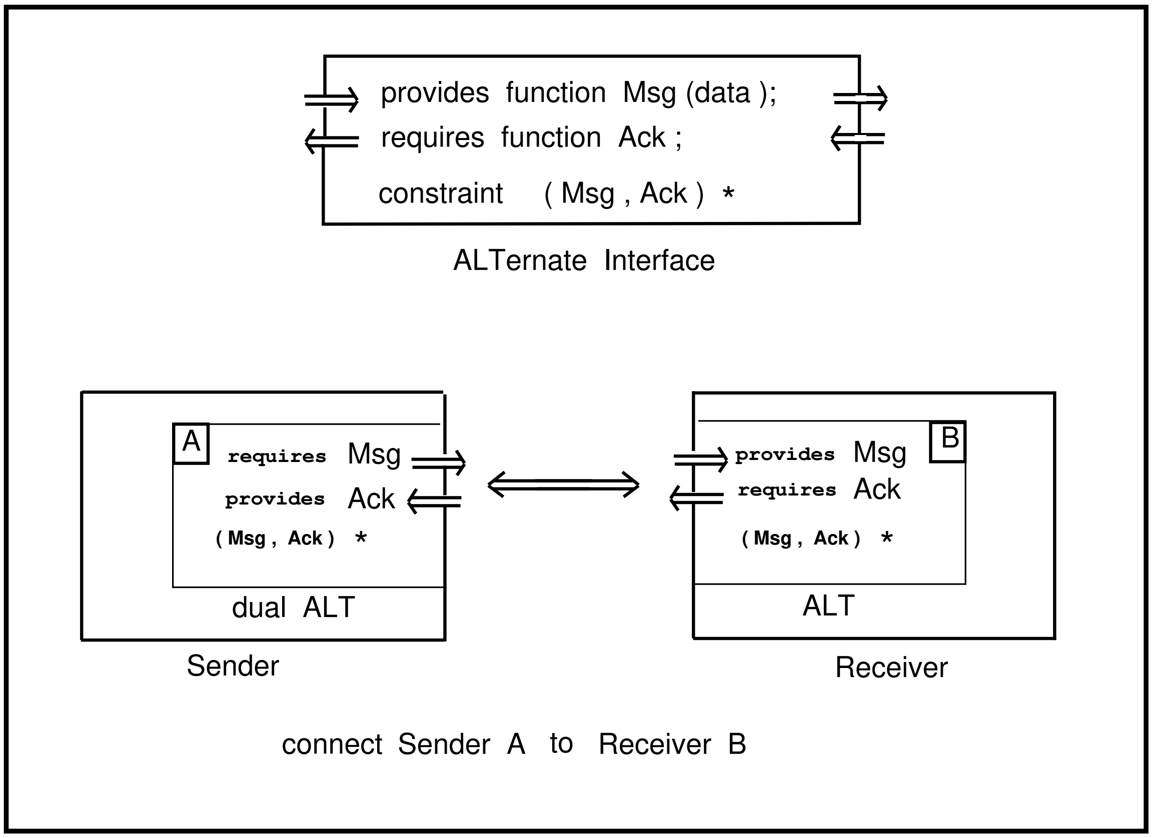

Figure 5: Dual services and a service connection between them

Examples

A *service* can be viewed as a *plug* and its dual as a matching *socket* (and conversely). This is a notion which is very common in the hardware world. For example, the RS-232 standard defines a set of twenty or so wires (features) together with their semantics, i.e., the direction of the signals on each wire and the protocol defining the orderings of the signals. An interface for a hardware component such as a modem or computer can provide a number of RS-232 ports, each one being declared either RS232 or *dual* RS232

[8]As we shall see, duality commutes with *provides* and *requires*.

(duality declares which *side* of the RS232 interface is provided, DTE or DCE). Then, a single ribbon cable can be strung between two such dual services provided by different pieces of equipment. A correct connection of the twenty or so wires is assured because both services conform to the same protocol constraint (see example, Section 9).

Figure 5 shows two interfaces, each containing a service. The Sender interface contains service A, and the Receiver interface contains service B. Service B is an instance of an interface called ALT. In this example features are functions and the dual of a provides feature is a requires feature with the same name, signature and constraints. Service A contains a dual of each feature of ALT together with the same constraint (explained later). A is therefore a dual service to B. Using service names to qualify feature names, the Receiver interface contains a *provides* function, B.Msg and a *requires* function, B.Ack; the Sender interface contains a *provides* function, A.Ack and a *requires* function, A.Msg.
□

Finally, we note here that an interface may have the capability to relate features that are specified in separate services within that interface. Constraints within the interface could be used, for example, to specify protocols involving two services: e.g. "whenever somebody contacts me on service A, then I report that contact on service B". So services in the same interface, although they group features together, are not necessarily disjoint as regards the behavior of components conforming to that interface.

9 Plug and Socket Architectures

Plug and socket architectures are interface connection architectures in which interfaces are allowed to provide services, and dual services may be connected together by a *service connection* :

service connection	=	a single connection between two dual services that denotes a set of interface connections, one between each pair of dual features in the two services.

One service connection denotes a number of individual connections between interface features. One may imagine in an ADL that a service connection has a concise notation using service names, so that it is notationally no more complex than a connection between a pair of interface features. One service connection therefore reduces the notation required for a set of interface connections in a large architecture to the same complexity as the notation for a single member of that set — rather like a mess of separate wires being bundled together into a single cable.

A service connection is correct if all of the interface connections it denotes are correct.

For many common examples of dual services, it is easy to show that the set of interface connections denoted by a service connection between those services must satisfy the service constraints. This is true for common kinds of protocols which constrain the order in which interface features are used, for example. In such cases, if two modules having the dual services conform to their interfaces, then they will communicate correctly on a connection between those services. We give an example below of correctness of a service connection between dual services with a simple protocol constraint.[9]

Examples

Correctness of a service connection

Figure 6 expresses the interfaces and connections shown in Figure 5. The constraint on the ALTernate interface describes a simple communication protocol (in this case using regular expressions). It should be read as saying the correct use of the interface is to call function Msg followed by a call to function Ack followed optionally by another call to Msg which must be followed by a call to Ack, ad infinitum.

[9]A general definition of the constraints for which service connections are known to be correct is beyond the scope of this paper.

```
class ALTernate is
provides
    function Msg(Data);
requires
    function  Ack;
constraint:
    (Msg; Ack)*
end class;

class Sender is
   service A : dual ALTernate;
end class;

class Receiver is
   service B :  ALTernate;
end class;

S: Sender; R: Receiver;

connect S.A to R.B;
```

Figure 6: A simple plug and socket architecture

In other words, there is an Ack following each Msg and a call to Msg may not happen until a preceding call to Ack (except for the first Msg, obviously).

The Sender and Receiver interfaces contain dual instances of services of type ALTernate. As long as each Sender and Receiver module conforms to the constraints in its interface (which includes the constraints in the services in its interface) then a connection between the two services will also be correct. To see this, consider that the Sender must obey the constraint in its service. Its functions have *provides/requires* modes that are duals of those in the ALTernate interface. Its constraint may be read as "Sender may call the Msg function (which it requires) but someone must call the Ack function (which it provides) before Msg can be called again." Similarly, the Receiver must obey the same constraint in its service. The constraint is over functions with the same *provides/requires* modes as the functions in the ALTernate interface, and may be read as "After the Msg function is called by someone, the Receiver must call the Ack function, and may not call it again until the Msg function is called again."

Now the semantics of interface connections is that a use of a required feature is replaced by a use of the provided feature connected to it (e.g., aliasing of function calls). So, if the Sender calls its interface Msg function then the Receiver's interface Msg function is called; similarly a call by the Receiver to its Ack function results in a call to the Sender's Ack. Therefore, the two interface constraints will be simultaneously satisfied if the modules conform to their interfaces. That is, "The Sender may call the Msg function after which the Receiver must call the Ack function. Only after the Ack function call may the Sender call the Msg function again." So, the communication defined by this service connection will satisfy the interface constraints, assuming interface conformance.

A plug and socket architecture for our compiler

By looking at the interfaces in Figure 4 one cannot tell which functions are intended to be connected. One cannot easily decide which other module is intended to use the Parser's Initialize function, or whether the Initialize and Semantize functions (in the Parser and Semanticizer) were expected to be used by the same module. If there is an error in the connections (very common in hardware modelling languages) there is no easy cross check. Indeed, we have used the same function names in different interfaces to suggest intended connections. In general it is difficult to tell which feature in one interface

```
class CodeGenerator_Semanticizer is
 provides:
   function Incremental_Semantize(Data : Tree; Context : Tree);
end class;

class CodeGenerator_Parser is
 provides:
   function Initialize();
 requires:
   function Generate(Data : Tree);
end class;

class Parser_Semanticizer is
 provides:
   function Semantize(Data : Tree);
 requires:
   function FileName() return String;
end class;
```

Figure 7: Interface services for compiler components

was intended to be connected to what other features in a different interface or which features in a particular interface were meant to be associated together for the purpose of connection — i.e., are part of a communication protocol.

The interface connection architecture for our compiler could be refined into a plug and socket architecture. Services (plugs and sockets) could be defined to group the features used to communicate between each pair of interfaces in the architecture. These services are shown in Figure 7. We have used obvious mnemonics to indicate which pairs of modules are intended to communicate by a given service. Using these services we can rewrite the interface connection architecture in Figure 4 as in Figure 8. Each interface contains two services, with each service encapsulating the communication between a specific pair of interfaces; each pair of interfaces contains duals of the service defining the communication between them. The three service connections are shown in Figure 8, and the architecture is depicted graphically in Figure 9.

Advantages of services are illustrated in this and the previous example. First an interface is structured into groups of features (the plugs and sockets) that indicate which other types of interfaces it should be connected to in architectures — i.e., those with dual services. So services indicate context with more precision than before. Secondly, service connections are less prone to trivial syntactic errors because they connect dual services – not simply individual features. Thirdly, as shown in the previous example, service connections can often result in correct sets of connections, and the correctness is easy to determine.

Refining communication in the compiler architecture

Services are also useful in the process of *refining* the detail in the communication between components of an architecture without increasing the complexity of its syntactic representation in an ADL. As the detail in specifying the communication between the components increases, the number of features in each of the services in Figure 7 increases. But while the plugs become more complex, the plug and socket architecture as shown in Figure 8 can remain unchanged.

For example, one could refine the service defining the communication between the Semanticizer and the Code_Generator. The Semanticizer could supply more a fine grained interface to the Code_Generator, allowing the Code_Generator to ask for specific symbol table information, or different semanticization schemes. The CodeGenerator_Semanticizer interface might become so complicated that we subdivide

```
class Parser is
 provides:
   C_P : service CodeGenerator_Parser;
   P_S : dual service Parser_Semanticizer;
 specification ...
end Parser;

class Semanticizer is
 provides:
   C_S : service CodeGenerator_Semanticizer;
   P_S : service Parser_Semanticizer;
 specification ...
end Semanticizer;

class Code_Generator is
 provides:
   C_P : dual service CodeGenerator_Parser;
   C_S : dual service CodeGenerator_Semanticizer;
 specification ...
end Code_Generator;

P: Parser; S: Semanticizer; G: Code_Generator

connect
  P.P_S to S.P_S;
  G.C_P to P.C_P;
  G.C_S to S.C_S;
```

Figure 8: A plug and socket architecture for a compiler.

its features even further into Symbol_Table features and Semanticize features. Such a refinement is illustrated in Figure 10. Similar expansions in detail could be made in the other services in our compiler architecture. The Parser interface could add features which select among different grammars (C, C++, etc). The Code_Generator could provide features which control its heuristics (fast generation of inefficient code, slow generation of efficient code) or its target instruction set (i486, Sparc, etc). The number of connections (and potential misconnections) in the interface connection architecture corresponding to such a plug and socket architecture becomes large.

The expanded CodeGenerator_Semanticizer service can be used in place of the previous service in the plug and socket architecture in Figure 8. The syntactic representation of the interfaces and the connections need not change, so the syntactic complexity of the architecture remains constant. Of course, the meaning of the interfaces and connections has changed. Whereas before each service connection denoted two or three function connections, the new service connections denote dozens or more function connections.

Adding new components to the compiler architecture

Services also provide a simple and error-free way to add new components to an architecture. Suppose the new component's interface is already designed. The dual of the new component's interface is added as a new service to each of the interfaces of those components that should be connected to the new one. Corresponding service connections between these new services and the new component interface are added to the architecture. The new features in interfaces are cleanly encapsulated in the new services,

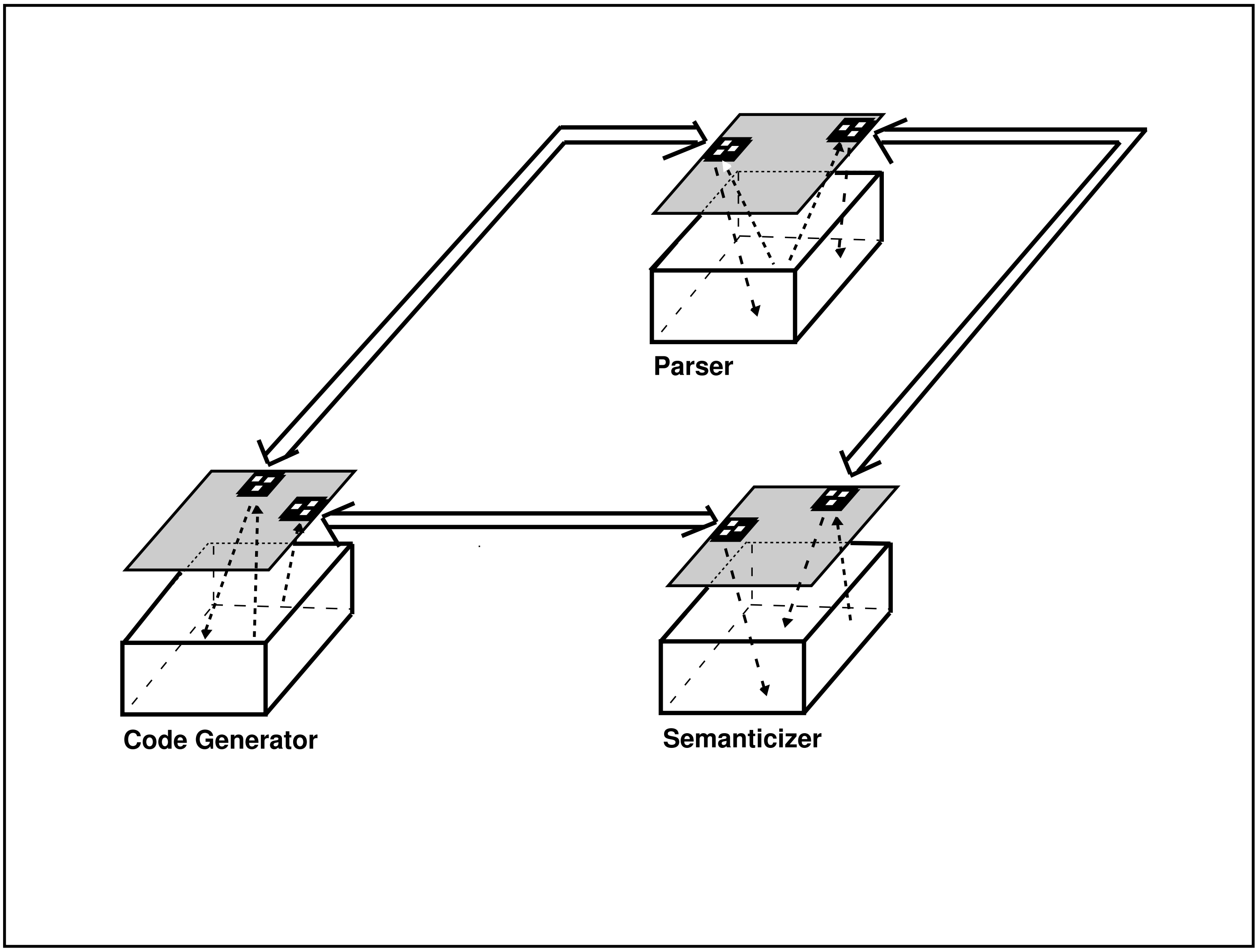

Figure 9: A plug and socket architecture and conforming system.

and separated from previous features, and the new connections are clearly defined. Mis-connecting new features with features in existing components rather than the new component is avoided.

For example, at some point in the development of an architecture communication integrity will become a concern. As mentioned in Section 6, all connections between more than one component and any unsafe component must be expressed in the architecture, otherwise a system is likely to have communication between its components that is not specified by the connections in the architecture.

Assume that each of the components in our compiler needs to perform I/O with the file system. A typical file system is certainly unsafe. The architecture in Figure 9 should be extended by adding a file system component and connections between it and the existing components. We may suppose that an interface for a file system, say something like Figure 11, is already in a component library. So, we add the dual of this interface as a new service in each of the interfaces for the Parser, Semanticizer, and Code_Generator. Now each component is specified to have a socket for connection to a file system. The architecture is then extended by adding a File_System interface and connections between it and these new services as shown in Figure 12. Assuming that each of the compiler components conforms to its new interface, which implies that they use the file system service correctly, then the new connections

```
class Symbol_Table is
 provides:
  function Is_Apply_Node(Data: Tree) return Boolean;
  function Is_Selection_Node(Data:Tree) return Boolean;
  function Is_Statement_Node(Data:Tree) return Boolean;
  ...
  function Line_Number(Data:Tree) return Integer;
  function Column_Number(Data:Tree) return Integer;
  function Attribute_Of(Data:Tree; Attribute_Index:Integer) return Attribute;
  ...
end class;

class Semanticize is
 provides:
  function Semantize_Compilation_Unit(Data:Tree; ST:Symbol_Table);
  function Semantize_Statement(Data:Tree; Context:Tree; ST:Symbol_Table);
  function Semantize_Expression(Data:Tree; Context:Tree; ST:Stmbol_Table);
  ...
end class

class CodeGenerator_Semanticizer is
 provides:
   ST: service Symbol_Table;
   S : service Semanticize;
end class;
```

Figure 10: Expanded CodeGenerator_Semanticizer service interface

```
class File_System is
   type File;
provides
   function Open_File(name: String) return File;
   function Close_File(f: File);
   function Write(f: File; data: String);
   function Read(f: File) return String;
   ...
end class;
```

Figure 11: File System Interface.

will be correct.

□

Service connections are not restricted to being one-to-one or static. We may have architectures in which one plug is connected to several sockets, and plug-to-socket connections may depend upon execution time variables.

10 Conclusion

This paper presented three concepts of *architecture*: *(i)* the *object connection architecture*, *(ii)* the *interface connection architecture*, and *(iii)* the *plug and socket architecture*. The first two kinds of

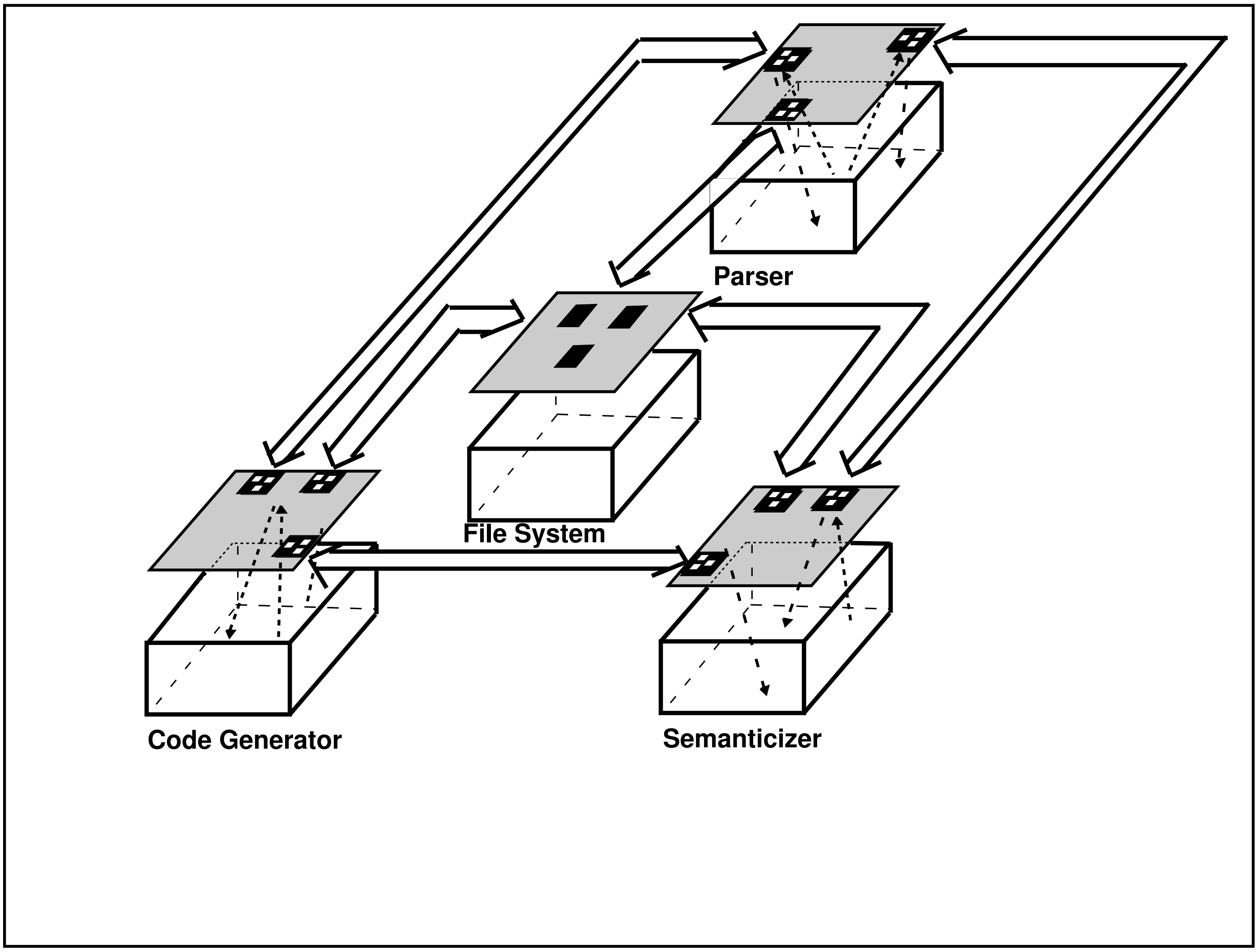

Figure 12: The compiler's plug and socket architecture with File System.

architecture differ in the way they are related to systems. An object connection architecture, typical of many object oriented systems, is defined by the system. Consequently, as illustrated in the paper, an object connection architecture cannot be used as a template to construct and modify systems, or to decide questions about the correctness of the system.

The interface connection architecture overcomes this problem by requiring that all communication into and out of a component go through that component's interface. A more complex concept of interface is needed to support interface connection architecture. This allows the communication architecture to be defined purely in terms of the interfaces, before components are constructed to implement those interfaces. We have indicated how interface connection architectures are templates for system development and modification.

Conformance of a system to an interface connection architecture requires both *interface conformance* and *communication integrity*. Conformance is often difficult to determine, and in general, undecidable. Also, interface connection architectures have the effect of raising the communication topology of the system to the level of the architecture, thus introducing concerns about scale early in the development process.

The final concept of architecture, the plug and socket architecture, preserves the advantages that the interface connection architecture has over the object connection architecture while introducing some ways of dealing with the scale and conformance problems of the interface connection architecture. Through the use of *services* and the notion of *duality*, an interface's specfication may be kept manageably compact. We have sketched how language syntax might take advantage of plug and socket architecture to reduce complexity, and we also indicated how services may be used to lessen the burden of proof for system/architecture conformance.

The examples we have used have all been rather simple, static architectures. However, the observations of this paper apply to dynamic architectures, where connections and components may be created or removed as during the actual execution of a software system.

Further works lies in determining more powerful and useful concepts of architecture, and in incorporating connection topology into the notion of architectural class (or type).

References

[1] R. Allen and D. Garlan. Formalizing architectural connection. In *Proceedings of Sixteenth International Conference on Software Engineering*, pages xx–yy. IEEE Computer Society Press, May 1994.

[2] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[4] D. Garlan and M. Shaw. *An Introduction to Software Architecture*, volume I. World Scientific Publishing Company, 1993.

[5] W. Wayt Gibbs. Software's chronic crisis. *Scientific American*, 271(3):86, September 1994.

[6] IEEE, Inc., 345 East 47th Street, New York, NY, 10017. *IEEE Standard VHDL Language Reference Manual*, March 1987. IEEE Standard 1076–1987.

[7] Intermetrics Inc., Cambridge, Mass. *Ada 9X Reference Manual*, June 1994. ANSI/ISO Draft International Standard.

[8] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek. Report on the programming language Euclid. *ACM SIGPLAN Notices*, 12(2), February 1977.

[9] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. 1995.

[10] David C. Luckham, James Vera, Doug Bryan, Larry Augustin, and Frank Belz. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

[11] K. Nygaard O-J. Dahl. Simula – an algol-based simulation language. *Comm. A.C.M.*, 9(9):671–678, 1966.

[12] D. L. Parnas. The influence of software structure on reliability. In *Proceedings of the International Conference on Reliable Software*, pages 358–362, April 1975.

[13] Alexandre Santoro and Woosang Park. SPARC-V9 architecture specification with Rapide. to appear, Stanford CSL Technical Report, 1995.

[14] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language*. Kluwer Academic Publishers, 1991.

[15] US Department of Defense, US Government Printing Office. *The Ada Programming Language Reference Manual*, February 1983. ANSI/MIL-STD-1815A-1983.