IEEE Transactions on Software Engineering, vol. 21, no. 4, 1995, pp. 314-335

Abstractions for Software Architecture and Tools to Support Them

Mary Shaw, Robert DeLine, Daniel V. Klein,
Theodore L. Ross, David M. Young, Gregory Zelesnik

Computer Science Department
Carnegie Mellon University
Pittsburgh PA
and various other current affiliations[1]

Version of March 8, 1995

Abstract

Architectures for software use rich abstractions and idioms to describe system components, the nature of interactions among the components, and the patterns that guide the composition of components into systems. These abstractions are higher-level than the elements usually supported by programming languages and tools. They capture packaging and interaction issues as well as computational functionality. Well-established (if informal) patterns guide architectural design of systems. We sketch a model for defining architectures and present an implementation of the basic level of that model. Our purpose is to support the abstractions used in practice by software designers. The implementation provides a testbed for experiments with a variety of system construction mechanisms. It distinguishes among different types of components and different ways these components can interact. It supports abstract interactions such as data flow and scheduling on the same footing as simple procedure call. It can express and check appropriate compatibility restrictions and configuration constraints. It accepts existing code as components, incurring no runtime overhead after initialization. It allows easy incorporation of specifications and associated analysis tools developed elsewhere. The implementation provides a base for extending the notation and validating the model.

Keywords: Software architecture, architecture description language, software system organization, architectural abstraction, software engineering

[1]Mary Shaw holds a joint appointment with the Software Engineering Institute, Carnegie Mellon University. Daniel V. Klein is currently with LoneWolf Systems, Pittsburgh PA. Theodore L. Ross is currently with Digital Equipment Corporation, Littleton MA. David M. Young is currently with Madeira Software, Inc., Beverly MA. Electronic mail contact address: mary.shaw@cs.cmu.edu

Table of Contents

1. Introduction

Software engineers often describe the "architectures" of their systems. These descriptions address high-level aspects of the systems such as the overall organization, the decomposition into components, the assignment of functionality to components, and the way the components interact. These architectural descriptions often use box-and-line diagrams and phrases such as "pipe-and-filter system" and "client-server model".

While architectural descriptions use high-level abstractions, the corresponding implementations are written in conventional programming languages. However, composing a system from subsystems is a substantially different activity from programming the underlying algorithms and data structures. Hardware designers confront the same situation; they recognize a number of distinct design levels, each with its own design issues, models, notations, componentry, and analysis techniques [Bell & Newell 71]. In the same way, different levels of software design require different kinds of components, different ways of composing components, different design issues, and different kinds of reasoning. The vocabulary gap between requirements and programming is substantial. Filling the gap requires better models and notations for the intermediate step. This is the goal of the emerging field of software architecture.

The *architecture of a software system* defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions. Architectural definitions can be composed to define larger systems. Elements are defined independently so they can be re-used in different contexts. The architecture establishes specifications for individual elements to be written in a conventional programming language. A number of commonly-used patterns, or idioms, are in widespread informal use; these architectural styles can be captured as general templates for families of related systems. This holds particular promise for domain-specific systems. [Garlan & Shaw 93, Mettala & Graham 92, Perry & Wolf 92]

Our primary considerations here are supporting architectural abstractions, localizing and codifying the ways components interact, and distinguishing among the various packagings of components that require different forms of interaction. Our focus is largely pragmatic. Our first concern has been to identify, classify, and support a variety of components and their connections. We will refine the notation and develop a formal base over time.

A sound basis for software architecture promises benefits for both development and maintenance. Design should benefit from improved abstractions, notations, and analysis. Architectural definitions should help provide good specifications for programming activities. Since the architectural definition also serves as the specification for system build, the specifications and code will be less likely to diverge. Maintenance should benefit in two ways. First, about half of maintenance effort is dedicated to understanding the system in preparation for making changes; explicit definition of the architecture should reduce this cost. Second, system architectures degrade over time; carrying the architectural definition into maintenance should reduce this tendency toward degradation.

Additional development benefits should accrue through better information to guide software reuse. Reuse is currently impeded by differences in component packaging. For example, a given functionality might be packaged as a procedure, a communicating process, an object, or a filter; it might interact with other components by calls, message-passing, method invocation, or shared data. These differences in packaging are recognized only informally and are not supported by programming languages and tools; neither formal nor informal guidance shows when and how to use them. As a result, it is often unclear whether components with compatible functionality will actually be able to interact properly. Documenting a system's structure and properties in a rigorous way has obvious advantages for maintenance. Much of the time spent on maintenance goes to understanding the existing code; this effort should be reduced substantially if the original design structure is captured clearly and explicitly. In addition, retaining the designer's intentions about system organization should help maintainers preserve the system's design integrity.

A growing community of researchers is focusing on software architecture. There has been long-standing interest in particular classes of architectures such as objects (focused by the OOPSLA conference), pipelines (focused by the USENIX conference), and client-server systems (a subject of intense attention in the commercial data processing arena). Early attention to the variety of idiomatic patterns of system organization and the possibility of organizing this knowledge systematically [Shaw 88, Perry & Wolf 92, Boehm & Scherlis 92, Garlan & Shaw 93] have led to enough activity to sustain workshops on architectural design [Barstow & Wolf 93], software design with strong participation from architecture [Lamb 95], patterns for design [PLoP 94], and interface definition languages [Wing 94]. Software architecture emerged as a key theme at a recent workshop on directions in software engineering [Habermann & Tichy 92]. A major effort to gain design power for specific kinds of problems addresses domain-specific software architectures (DSSAs) in several specific domains [Mettala & Graham 92, Hayes-Roth & Tracz 93]. Formalizations are beginning to emerge [Allen & Garlan 94a,b]. As a natural consequence, languages for describing architectures are emerging [Dean & Cordy 93, Rapide 93].

The need for a notation to describe how subsystems written in typical programming languages connect to form larger systems is not a new concern. In 1975, DeRemer and Kron [DeRemer & Kron 76] argued that creating program modules and connecting them together to form larger structures were distinct design efforts; they created the first module interconnection language (MIL) to support the connection effort. In an MIL notation, modules import and export resources, which are named elements such as type definitions, constants, variables, and functions. Compilers for MILs ensure system integrity with intermodule type checking: they check that if one module uses a resource that another provides, the types of the resources match; that if a module declares it provides a resource, it actually does; that if a module uses a resource, it has access to that resource; and so on. Since DeRemer and Kron's MIL, MILs have been developed for specific languages, like Mesa [Mitchell et al 79] and Ada [Campos & Estrin 78], and have provided a base from which to support software construction [Thomas 76], version control [Cooprider 79], system families [Tichy 79], and dynamic configuration [Magee et al 89]. Enough examples are available to develop models of the design space [Perry 87, Prieto-Diaz & Neighbors 86].

These early module interconnection languages require considerable prior agreement between the developers of different modules. For example, they assume that simple name matching can be used to infer inter-module interaction, that all modules are written in the same language, that all modules are available during system construction, and that module interfaces describe the other modules with which they interact. Newer work has begun to soften these restrictions. In the Darwin language, modules can be dynamically instantiated and bound at runtime [Magee et al 93]. Polygen [Callahan & Purtilo 91] augments a module interconnection language with an inference engine that deduces from a user-defined set of rules how (or whether) a system can be integrated from set of modules.

These modules can be implemented in multiple programming languages, and the machinery needed to connect them can be richer than the usual procedure linkage, for example, a software bus [Purtilo 90]. This kind of system requires expanding the notion of a MIL to include specifics about a module's implementation, such as its programming language, its hardware/operating system platform, and the communication media needed to access it; the resulting richer notation has been termed a module interconnection formalism (MIF). To build truly composable systems we must allow flexible, high-level connections between existing systems in ways not foreseen by their original developers. Essentially independently, developers of "open" software products have designed interchange representations such as PICT (line drawings), RTF (formatted text), SYLK (spreadsheet layouts), and MIF (formatted text) to allow distinct products to interact by data interchange. Although these were originally static, newer developments such as CORBA, OLE, and OpenDoc (all for objects) support dynamic sharing.

Systems often exhibit an overall style that follows a well-recognized, though informal, idiomatic pattern. Garlan and Shaw survey half a dozen of these patterns and illustrate their use in case studies [Garlan & Shaw 93]. They identify pipes and filters, data abstractions or objects, implicit invocation, hierarchical layers, repositories, and interpreters as a useful (though incomplete) set of well-known patterns, or styles, of system organization. These styles differ both in the kinds of components they use and in the way those components interact with each other. As advocates of various of these architectures explain, adherence to the rules of the style enhances both software development and subsequent maintenance.[2] The rules of the style usually restrict how to package components—e.g., as procedures, objects, or filters. As a result, components may not be usable in all styles; code may not reusable because its interface makes incompatible assumptions. In Unix, for example, the functionality of "sort" is available both in the form of a filter and in the form of a procedure.

The remainder of this paper is organized as follows: Section 2 introduces our model of software architecture and its notation; Section 3 describes UniCon, an initial language for implementing the model; Section 4 highlights its implementation; Section 5 describes our experience with UniCon.

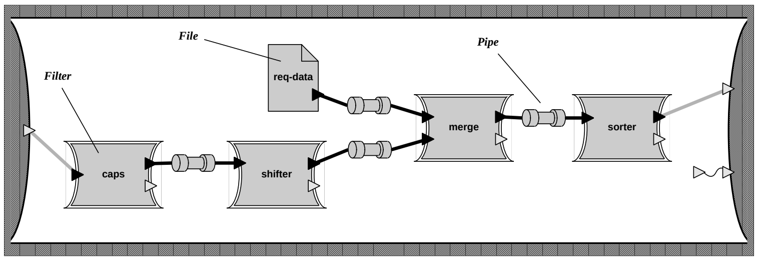
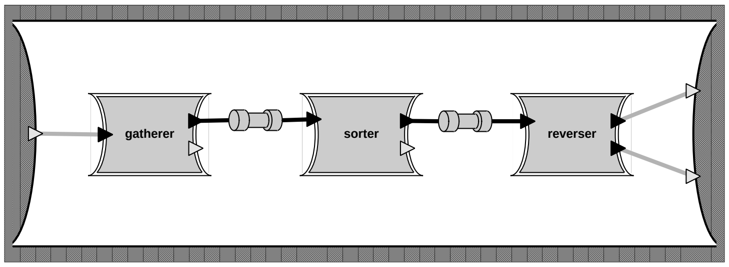


Figure 1. Pipe-and-filter example: the KWIC indexer.

We will use three examples throughout the paper. Although architecture diagrams often do not make visual distinctions among the interactions they depict, here we use different markings for different types of connections. The first example, shown in Figure 1, is a Unix-style pipe-and-filter system built from both filters and files, with the wrinkle that the pipeline merges two streams—a configuration that is difficult or impossible to describe in most shells. The system implements a KWIC (keyword in context) indexer; we have used a very similar task as a class exercise in a software ar-

[2]However, the advocates often neglect to mention that different architectures are appropriate in different situations. Choosing the most appropriate architecture for a given problem (or domain) remains an open problem.

chitecture class [Garlan & Shaw 94]. The challenge of this example is to make complex topologies as easy to describe as simple ones.



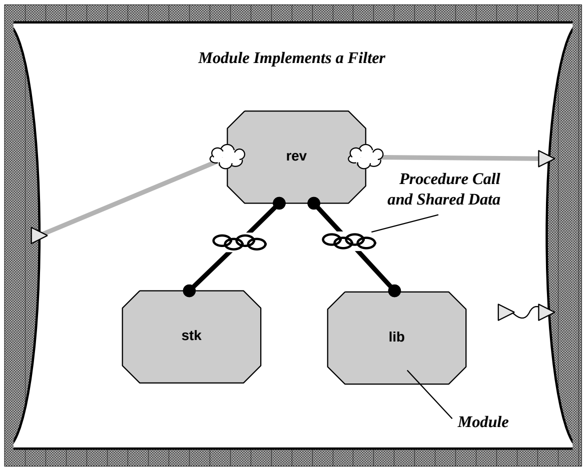


Figure 2. Heterogeneous implementation of a pipeline using both pipes and procedures. The upper diagram is the high-level view of the system, a simple pipeline. The lower diagram shows the implementation of the right-most component in the upper diagram.

The second example, shown in Figure 2, combines a pipe-and-filter architecture with a conventional procedural implementation of one of the filters. The challenge of this example is to compose architectural descriptions and to establish the correspondence between the abstraction of a pipeline and its implementation as calls on system procedures.

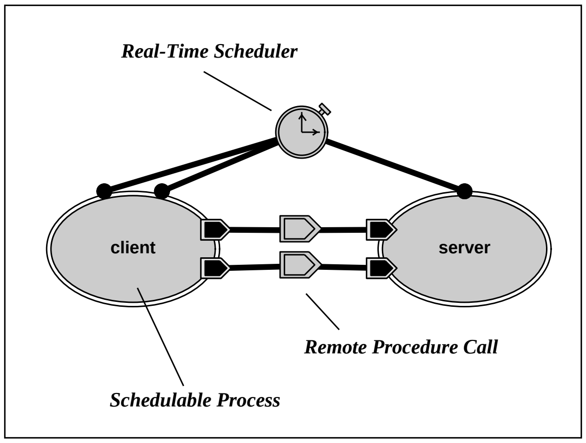


Figure 3. Real-time client-server system with two schedulable tasks sharing a computing resource. The tasks also interact via remote procedure call.

The third example, shown in Figure 3, involves coordinating real-time tasks. A simple periodic real-time system has a number of tasks that must run on specified schedules. More challenging scheduling problems arise when tasks interact. We use an example with two schedulable tasks that interact through remote procedure calls. Based on a system timer, the client must periodically perform computations which involve calling upon services provided by the server; correctness requires that these events take place at specific times. The challenge of this example is to incorporate an external analysis tool that determines the legality of the configuration (especially to make guarantees about schedulability) and to convert the code for the tasks into schedulable processes that run on a real-time operating system.

2. *Model and Notation*

This section describes an informal model for an architectural description language. The language is intended to aid designers in first defining software architectures using abstractions they find useful and then making a smooth transition to code. Our long-term objective is to fully elaborate this model and to support it with notation and tools. Currently, we are more strongly motivated by practical utility of the model than by formal foundations. At present, the model provides a framework for understanding our initial implementation, which is described in Section 3.

The model addresses several issues in novel ways:

- It supports abstraction idioms commonly used by designers—for example, explicitly distinguishing different types of elements and providing type-specific analysis support.
- It specifies packaging properties as well as functional properties of components—for example, distinguishing clearly between functionality delivered in the form of a filter from functionality delivered in the form of a procedure.
- It provides an explicit, localized home, called a connector, for information about the rules for component interactions, such as protocols, interchange representations, and specifications of data formats for communication.
- It defines an abstraction function to map from code or lower-level constructs to higher-level constructs. This is similar to Hoare's technique for abstract data types [Hoare 72].
- It is open with respect to externally-developed construction and analysis tools. It supports collection and delivery of relevant information to tool and return of results from the tool.

Software system composition involves different tasks from writing modules: the system designer defines roles and relationships rather than algorithms and data structures. These concerns are sufficiently different to require a separate language. The architectural language must support system configuration, independence of entities (hence reusability), abstraction, and analysis of properties ranging from functionality to security and reliability [Shaw & Garlan 93]. The model must be supported by a notation.

2.1. Components and Connectors

Systems are composed from identifiable *components* and *connectors* of various distinct types. The components interact in identifiable, distinct ways. Components roughly correspond to compilation units of conventional programming languages and other user-level objects such as files. Connectors mediate interactions among components; that is, they establish the rules that govern component interaction and specify any auxiliary implementation mechanism required. Connectors do not in general correspond individually to compilation units; they manifest themselves as table entries, buffers, instructions to a linker, dynamic data structures, sequences of system calls embedded in code, initialization parameters, servers that support multiple independent connections, and the like. During system design, it is important to work with good abstractions for interaction without concern for whether their implementations are localized. It is helpful to think of the connector as defining a set of *roles* that specific named entities of the components must *play*.

Our model thus describes software systems in terms of two kinds of distinct, identifiable elements: components and connectors. Each of the two elements has a type, a specification, and an implementation. The specification defines the units of association used in system composition; the implementation can be primitive or composite. Figure 4 suggests the essential character of the model.

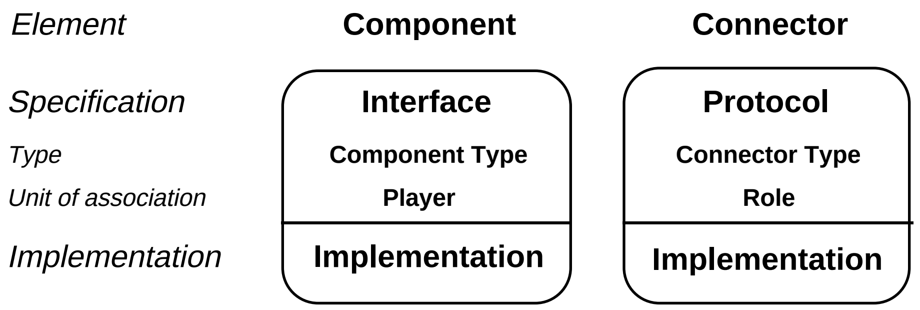


Figure 4: Gross structure of an architecture language.

Components are the locus of computation and state. Each component has an interface specification that defines its properties. These properties include the component's type or subtype (e.g. filter, process, server, data storage), functionality, guarantees about global invariants, performance characteristics, and so on. The specific named entities visible in a component’s interface are its players. The interface includes the signature, functionality, and interaction properties of its players.

Connectors are the locus of definition for relations among components. They mediate interactions but are not “things” to be “hooked up;” rather, they provide the rules for hooking-up. Each connector has a protocol specification that defines its properties. These properties include its type or subtype (e.g. remote procedure call, pipeline, broadcast, shared data representation, document exchange standard, event), rules about the types of interfaces it works with, assurances about the interaction, commitments about the interaction such as ordering or performance, and so on. The specific named entities visible in a connector’s protocol are roles to be satisfied. The interface includes rules about the players that can match each role, together with other interaction properties.

Components may be either primitive or composite. Primitive components may be implemented as code in a conventional programming language, shell scripts of the operating system, software developed in an application such as a spreadsheet, or other means external to the architectural description language. Composite components define configurations in a notation independent of conventional programming languages. This notation must be able to identify the constituent components and connectors, match the players of components with roles of connectors, and check that the resulting compositions satisfy the specifications of both the components’ interfaces and the connectors’ protocols.

Similarly, connectors may be either primitive or composite. They are of many kinds: shared data representations, remote procedure calls, data flow, document exchange standards, standardized network protocols, etc. The connectors derived directly from programming languages are typically asymmetric with two complementary roles: a procedure has a definer and multiple callers; data has an owner and multiple users; a simple data stream has one producer and one consumer. However, other, more abstract connectors may be symmetric, and their roles may be more numerous and more specific: in broadcast communication, all participants may be alike; in some event systems any component may raise or respond to any event. The usual import/export or provides/requires relation is too restrictive to express the relations used in practice—it fails to expose important distinctions. If rich, abstract connectors are to be defined, the architectural notation must be able to express complex interaction properties. These might be as diverse as

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

Primitive connectors may be implemented in a number of ways: as built-in mechanisms of programming languages (e.g., procedure calls or shared data); as system functions of the operating system (e.g., certain kinds of message passing); as library code in conventional programming languages (e.g., X/Motif); as shared data (e.g., Fortran `COMMON` or Jovial `COMPOOL`); as entries in task or routing tables; as a combination of library procedures and a single independent process for the connector (e.g., certain kinds of communication services); as interchange formats for static data (e.g., RTF); as initialization parameters (e.g., event period and process priority in a real-time operating system) and probably in a variety of other ways. Composite connectors may also appear in these diverse forms; we need (but do not yet have) ways to define them as well.

The example of Figure 2, heterogeneous implementation of a pipeline, uses different types of components and connectors. It also shows a composite implementation of one of the filters.

The remainder of this section deals with three issues of particular interest for general-purpose architectural tools: abstraction and encapsulation (Section 2.2), the appropriate analog for types (Section 2.3), and the ability to provide access to externally-developed tools (Section 2.4).

2.2. Abstraction and Encapsulation

For a composite element, the implementation part consists of

- a parts list (components and connectors)
- composition instructions (association between roles and players)
- abstraction mapping (relation between internal players and players of the composite)
- other related specifications (detailed properties of the parts and compositions).

This localizes and encapsulates information about the system structure rather than distributing it around the system in import/export statements. Since composition information is localized, global properties such as restrictions on topology or types of elements may be checked. Since the abstraction mapping is explicit, we have the opportunity to allocate responsibility for the code correctly implementing higher-level connectors. Further, the composition instructions make the matching of players and roles explicit, so we can break free of name matching as the sole means of making connections, as shown in Section 3.3.3, Program 4.

Common system-composition idioms, such as pipeline, client-server, or blackboard, can be defined as idiomatic patterns, or *styles*, of components and connectors. These patterns describe the types of components and connectors that can be used and may constrain the interconnection topologies. Indeed, some such styles (e.g., pipe-and-filter) are described primarily in terms of the prescribed form for communication, data sharing, or other interaction. In practice, the rules for these styles are usually implicit. The combination of localized definitions and higher-level elements makes it possible to formalize rules for styles [Abowd et al 93, Allen & Garlan 94a].

Abstract data types rely on an *abstraction function* to show the correspondence between the internal representation of a type and the abstract view that the user (and the specification) takes [Hoare 72]. For software architectures, abstractions are required to implement higher-level components in terms of lower-level ones. When a component or connector is not directly implemented by a programming language, its definition must explain how the abstract properties will be implemented. This might take the form of a manual or informal guidance. Even better, it could be a code template, an automated generator, or a formalization. No matter how it is represented, the definition must set out the programmer's responsibility. When a number of components are connected to form a larger component, the players of the defined unit may be more abstract than the players of the implementation. In that case, the definition must explicitly indicate the correspondence among one or more external player (abstraction), one or more players of the constituent components, and the implementation rule. This is the abstraction mapping of the component. Similarly, abstraction mappings will be required for composite connectors. This differs from data abstraction chiefly in that it maps not only data to an abstract value but rather data plus functions to a set of abstract player types.

2.3. Types and Type Checking

A problem similar to type checking in a programming language arises at four points in an architectural language. Two of these appear in the preceding discussion: the types of components and of connectors and their use in showing adherence to a style. As with any type system, types for components and connectors express the designer's intent about how to use the element properly and are most useful when the language checks them. The types for connectors and components are not merely enumerations of unrelated items; some are closely related to others. Architectural types describe expected capabilities and limit both what can appear in the construct's specification and the legitimate ways to use the construct. Examination of real systems shows that type hierarchies of this sort are useful. For example, there are many kinds of memories (components) and many kinds of event systems (connectors). Defining type structures for these elements requires the creation of taxonomies to catalog and structure the type variations. This is part of establishing a full model for architectural composition.

The third place where type checking appears is at the actual point of associating a component's players with a connector's roles. Each of the named entities in the interfaces and protocols must have enough type information and other specification to determine whether the connector definition allows the components to be associated as requested. Furthermore, a component may be used differently by different kinds of connectors. For example, a client might be indifferent to whether its servers are dedicated, shared, or distributed. An abstract pipe may be able to connect both filters and files (but not processes that share data directly). We must therefore support flexible associations between players and roles. For connectors such as procedure call that correspond directly to language constructs, this corresponds to the checks a linker may make. However, for richer, more abstract connectors the checks are more sophisticated.

The fourth opportunity for type-like checking arises when more than one architectural style is used in designing a single system. This problem also resembles that of reconciling multiple views. A comparison of architectures for a single problem [Shaw 95] explores this issue.

Components and connectors must be reusable in different settings, so it's important to deal reasonably with associations that are slightly mismatched. Common examples include mismatches between the order and types of a library procedure's arguments and those of the procedure intended to call it; remapping data formats to support sharing; and the subtle differences between remote procedure calls and local procedure calls. If component's packaging fails to match the packaging needed in use, mechanical adaptation may be possible. Our current implementation makes initial steps toward supporting adaptation in the face of mismatch.

2.4. Accommodating Analysis Tools

Architectural descriptions should be "open" with respect to analysis tools. We must accommodate techniques that are applied at the systems level of design. These analysis tools will often be developed independent of the model. They may address such properties as functional correctness, performance, and timing (e.g., allowable order of operations, real-time guarantees). The architectural description language should be able to interact with any analysis technique that works with information in the architectural specifications. It should be able to record the system-level specifications required by external tools as uninterpreted expressions, deliver information to the tools, receive results from the tools, and incorporate those results in the architectural description.

Perhaps the most natural kind of system-level analysis is that of functional specification, which might use pre- and post-conditions to check procedure calls, for example. Perry suggested this as part of his software interconnection model [Perry 87]. Rather than building a theorem prover into the system, the system could collect the assertions from a procedure's definer and a potential caller and invoke an external theorem prover to decide whether to allow the call. This will provide a much stronger check than either name matching or signature comparison.

A second example is rigorous analysis of the real-time properties of a system. For certain classes of systems, correctness depends on the time at which computations are completed, not just on whether the computations themselves are correct. The designer of a system must account for the computation times of individual modules as well as the complex interactions within the composition of modules. As described in Section 3.5, our implementation now supports rate monotonic analysis (RMA), which is a body of techniques for analyzing the schedulability of preemptive, fixed-priority systems. The Software Engineering Institute at Carnegie Mellon has developed a handbook for this family of analysis techniques [Klein et al 93].

3. *UniCon: Language for Universal Connector Support*

In order to gain experience with the practical details of an architectural description language, we have implemented a simplified initial system. The system, called *UniCon*, emphasizes the structural aspects of software architecture. It is higher-level and more general than existing mechanisms for system composition, but it is low-level compared to the model of Section 2. Specifically, the objectives of this implementation were the following:

- Address real problems of system description and composition; provide a prototype of a practical tool.
- Provide uniform access to a wide range of connection mechanisms. Select connectors that are available in the local computing environment so we can concentrate on providing them, not inventing them; however, select with diversity in mind.
- Help software designers to discriminate among different types of components and different types of connectors and to check the legitimacy of proposed configurations.
- Support both graphical and textual notations with interchange between the representations.
- Support analysis tools and specification notations developed by others. Preserve compatibility with programming tools in common use.
- Accept existing components. Use components written in ordinary programming languages, including those not written for use with this tool.
- Keep added run-time costs to a minimum. Ideally all UniCon-specific elements disappear by the time the executing system is initialized.

To speed the process of gaining experience, some simplifications were made to the model when implementing UniCon, namely:

- UniCon supports composite components but not composite connectors.
- Abstractions for components and connectors are built in, but new ones cannot yet be defined. These built-in elements sample a diverse space, but the set is in no sense complete and a unifying taxonomy is not yet provided.
- The only primitive components at present are compilation units. Although the implementation is largely language-indifferent, C is the language supported at present.
- The syntax has not been refined for conciseness yet. It can be a bit wordy, especially when making intermodule connections of procedures and data with no change of name.

This section begins by presenting the semantics of UniCon (Section 3.1). As an aid to intuition, this section anticipates the syntax discussion by providing a sample of the textual form of the KWIC example from Figure 1. The following two sections describe graphical syntax (Section 3.2) and textual syntax (Section 3.3); these are equivalent. We then describe the way existing code is incorporated as primitive components (Section 3.4) and the technique for using external analysis tools (Section 3.5).

3.1. Semantics

Following the model presented in Section 2, UniCon is based on two complementary kinds of constructs: the component and the connector. Their structures are symmetric, except that composite connectors are not yet supported. Each has:

- a Name
- a specification (called an Interface for a component, a Protocol for a connector)
- a component or connector Type
- a set of global assertions in the form of a Property List
- a collection of association units: Players for components, Roles for connectors. Specific details about these are specified in Property Lists.
- an Implementation

Each of these definitions can be understood solely in terms of its specification, and the properties of a system should be derivable from the specification of its components and connectors. This is closely related to Lam and Shankar's properties separability and composability [Lam & Shankar 94]. Each provides a template that is instantiated when it is used. Information about distinctions among components, connectors, players, and roles is used for an analog of type checking. The definition of language semantics is organized around this structure. Section 3.1.1 describes components, Section 3.1.2 describes connectors, and Section 3.1.3 describes type checking.

3.1.1. Components

Components define computational capabilities. A component consists of an interface that specifies the capabilities the component exports and an implementation, which may be primitive or composite. The interface of a component must be consistent with its implementation. In the case of a primitive implementation, such as a source file or executable, this is the responsibility of the programmer. In the case of a composite implementation, the interface must be consistent with the interfaces of its member components and the protocols of its connectors; its semantics should be derivable from the interfaces of its constituents.

The interface defines the computational commitments the component can make and constraints on the way the component is to be used. The interface provides the guarantees that will hold of the behavior and performance of the component. It should be possible to use the component by reference to the interface alone. The interface must include

- the component type
- assertions and constraints that apply to the entire component
- the players defined by the component, which each consist of a name and type and optional attributes like signature, functional specifications, constraints on use, or information required specifically by a component type (e.g., port bindings of Unix file descriptors).

A component type expresses the designer's intention about the general class of functionality to be provided by the component; it restricts the numbers, types, and specifications of the Players defined by the component. The players, which form the bulk of the interface, are the visible semantic units through which the component can interact, request and provide services, or be influenced by external state or events. The interactions are mediated by the roles of connectors. The detailed specifications of the players appear in the form of property lists, which are lists of attributes and their associated values. Program 1 shows an example of the textual description of both a primitive and a composite component, which will be referenced throughout this section. Component types and their associated players are discussed below in Section 3.1.1.1.

To provide some specific intuition for the semantics, we will use the pipeline implementation of KWIC presented in Figure 1. This example's pipeline involves four filters, one file, four pipes, and the two streams that are Players in the composed system. Three of the filters in this example have the canonical interface for a simple filter, specifying stdin, stdout, and stderr; merge, of course, has two inputs. For these filters the specification further indicates that the data on the stream is line-oriented. Program 1 shows the textual notation for two components from the KWIC example, which will be referenced throughout this section.

Component KWIC from Program 1 is an example of a system. A system is a component, usually a composite component, that is capable of independent operation in the context of a computer and its operating and runtime systems. The external specification of the system is the interface of the component. The system remains eligible for use as a component; indeed, the essence of systems integration is finding ways to treat as subsystems today those things that were systems last week. A system may interact with users and other systems. A system is closed in the sense that all the *players* of its *interface* are bound to its execution environment; it implements a function that is discrete and complete in the mind of its designer. This is the result of an architectural definition.

```
COMPONENT sort
   INTERFACE IS
      TYPE Filter
      PLAYER input IS StreamIn
         SIGNATURE ("line")
         PORTBINDING (stdin)
         END input
      PLAYER output IS StreamOut
         SIGNATURE ("line")
         PORTBINDING (stdout)
         END output
      PLAYER error IS StreamOut
         SIGNATURE ("line")
         PORTBINDING (stderr)
         END error
      END INTERFACE

   IMPLEMENTATION IS
      VARIANT sort IN "sort"
         IMPLTYPE (Executable)
         INITACTUALS ("-f")
      END sort
   END IMPLEMENTATION
END sort
```

```
COMPONENT KWIC
   INTERFACE IS
      TYPE Filter
      PLAYER input IS StreamIn
         SIGNATURE ("line")
         PORTBINDING (stdin)
         END input
      PLAYER output IS StreamOut
         SIGNATURE ("line")
         PORTBINDING (stdout)
         END output
      PLAYER error IS StreamOut
         SIGNATURE ("line")
         PORTBINDING (stderr)
         END error
      END INTERFACE

   IMPLEMENTATION IS
/* First instantiate the parts to use */
      USES caps INTERFACE upcase
      USES shifter INTERFACE cshift
      USES req-data INTERFACE const-data
      USES merge INTERFACE converge
      USES sorter INTERFACE sort

      USES P PROTOCOL Unix-pipe
      USES Q PROTOCOL Unix-pipe
      USES R PROTOCOL Unix-pipe

/* Associate players of some parts to players
   of interface. By default, all stderrs are
   bound to the external stderr */
      BIND input TO caps.input
      BIND output TO sorter.output

/* Describe the way KWIC is built from parts
   by directly associating players with
   roles */
      CONNECT caps.output TO P.source

      CONNECT shifter.input TO P.sink
      CONNECT shifter.output TO Q.source

      CONNECT req-data.read TO R.source

      CONNECT merge.in1 TO R.sink
      CONNECT merge.in2 TO Q.sink

/* Syntactic sugar for full connections */
      ESTABLISH Unix-pipe WITH
         merge.output AS source
         sorter.input AS sink
         END Unix-pipe
   END IMPLEMENTATION
END KWIC
```

Program 1. A primitive and a composite component.

A component's *implementation* may be either primitive or composite. Primitive implementations provide one or more ways to locate a definition in some programming language and are discussed below in Section 3.1.1.2. Composite implementations instantiate a set of components and configure them with connectors and are discussed in Section 3.1.1.3.

ComponentType	*Player Types supported*
Module	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, PLBundle, ReadFile, WriteFile
Computation	RoutineDef, RoutineCall, GlobalDataUse, PLBundle
SharedData	GlobalDataDef, GlobalDataUse, PLBundle
SeqFile	ReadNext, WriteNext
Filter	StreamIn, StreamOut
Process	RPCDef, RPCCall
SchedProcess	RPCDef, RPCCall, RTLoad
General	All (that is, any player type is allowed)

Table 1. Built-in component types and their players.

3.1.1.1. Built-in Component Types

Component types and player types are currently defined by enumeration, but extensions may be supported some day. Table 1 lists the component types currently supported and the players allowed for each. This set of component types was selected opportunistically: we wanted to reflect components and connectors used in practice and to cover as wide a variety as possible. Each of these is described in detail below.

- Component type *Module* corresponds to a compilation unit in a typical programming language. The *RoutineDef* players correspond to exported procedures and functions. The *RoutineCall* players correspond to imported procedures and functions. Similarly, *GlobalDataDef* and *GlobalDataUse* players correspond to import and export of named data. The *ReadFile* and *WriteFile* players provide an input/output capability.[3] These players correspond directly to language constructs or system calls. However, modules frequently define one or more coherent collections of players; when designers think about the architecture of the system, they think about the use of a collection of players rather than about all the individuals. UniCon captures this with an abstract player, *PLBundle*, that corresponds to a set of individual players related to procedure calls or data use. Modules are intended to provide definitions that will be linked in a single name space.
- Component type *Computation* is a specialization of *Module* whose interface is restricted to defining procedures, functions, and bundles thereof and using procedures, functions, and data defined elsewhere. It is intended to capture purely computational units that are collections of procedure definitions and calls. Similarly, component type *SharedData* is a specialization of *Module* whose interface is restricted to defining and referencing data.
- Component type *SeqFile* corresponds to sequential files in which lines, characters, or records (as specified by the *RecordFormat* attribute) are read sequentially from the front (*ReadNext*) and written sequentially at the end (*WriteNext*).

[3]This currently makes no provision for interactive input and output. This will be addressed soon by adding a connector for typescripts, later by adding a connector that provides protocol for interaction with a user-defined graphical interface.

- Component type *Filter* corresponds to Unix filters in which input arrives in streams and output is produced in streams (*StreamIn* and *StreamOut*). The syntax of the stream (the structure imposed on elements in the stream) may be specified by the *Signature* attribute.
- Component type *Process* corresponds to an independently scheduled process at the operating system level. It differs from *Filter* in that it interacts via remote procedure calls (*RPCDef* and *RPCCall*) rather than data flow.[4] *SchedProcess* is a special type of Process that admits of real-time analysis and scheduling; (see Section 3.5). A SchedProcess provides an abstract player *RTLoad* that provides the information required for real-time scheduling using attributes *SegmentSet, Trigger, SegmentDef*, and *TriggerDef*.
- A *General* component type allows any player types, thus allowing the definition of arbitrary components; it does not support analysis or checking. Using *general* components when it is possible to use more specific types defeats the purpose of component typing.

[4]Interprocess data sharing will be added in the future.

Attribute	*Req/Opt*	*Merge Rule*	*Applies to Components*	*Value / Default / Syntax*
InstFormals	optional	merge	all	Formal parameter list for instantiation of component. Default is no instantiation parameters. Syntax depends on rules of implementation language.
Variant	optional	replace	all	Used during instantiation of the component to select among variant implementations. Default is the first implementation provided. Syntax is name of a variant.
RecordFormat	optional	replace	SeqFile	Format of records in the file. Default is lines separated by <CR>. Syntax depends on rules of implementation language.
Library	optional	replace	Computation, Module, SeqFile, SharedData	Changes default value of MinAssocs to 0 (i.e., allows unused players). Also provides hint to builder to generate library archive rather than simple executable form.
EntryPoint	optional	replace	Module, Computation, Process, SchedProc	Point at which to start execution. Default is main program of module. Syntax is procedure name.
Priority	optional	replace	SchedProc	Priority at which real-time operating system should run the process. Default is highest. Syntax is integer.
Processor	optional	replace	All	Processor on which the component will be executable. Default is local processor. Syntax is processor name.
SegmentDef	required	replace	SchedProc	Definition of a segment of code in the implementation of a real-time component. Syntax is name followed by ‘;’ followed by execution time in seconds. No default
TriggerDef	optional	replace	SchedProc	Definition of external stimulus that activates a segment. Syntax is name followed by ‘;’ followed by period of stimulus in seconds. No default.
RPCTypesIn	optional	merge	Process, SchedProc	Auxiliary information required to derive language-independent type information for RPC generator.
RPCTypedef	optional	replace	Process, SchedProc	Auxiliary information required to derive language-independent type information for RPC generator.

Table 2. Attributes that apply to components.

Attributes in property lists provide further specification of a component. Each attribute is relevant to one or more component types and may be required or optional; it must specify what to do if multiple values are specified for the attribute, for example, in different property lists. The possibilities are currently to use the most recent (replace), to construct a list of all values (merge), and to refuse multiple definitions (error). Certain attributes pertain to a component as a whole. For example, in cases where it is possible to provide instantiation parameters to an entire component, those are provided with the property list for the component itself. Table 2 lists component attributes, and Table 3 gives the attributes defined for Players. They are largely specific to the type of the Player.

Attribute	*Req/Opt*	*Merge Rule*	*Applies to Players*	*Value / Default / Syntax*
MaxAssocs	optional	replace	all	Maximum number of associations the player can be involved in. Default is unlimited. Syntax is integer.
MinAssocs	optional	replace	all	Minimum number of associations the player can be involved in. Default is 1. Syntax is integer.
Signature	required	error	RoutineDef, RoutineCall, GlobalDataDef, GlobalDataUse, StreamIn, StreamOut, ReadFile, WriteFile, RPCDef, RPCCall	Formal parameter list or type definition, possibly with default values for the parameters. No default value; this is such an essential part of the definition of these players that even an empty parameter list should be explicitly indicated. Syntax is dependent on type of player and implementation language.
Portbinding	required	replace	StreamIn, StreamOut	Binding of port for pipe. Defaults are stdin for StreamIn and stdout for StreamOut. Syntax is integer or port id such as stdin, stdout, stderr.
Member	required	replace	PLBundle	Defines a player to be a member of an abstract collection of players that correspond to programming language constructs procedure and data. Syntax is name, player type, and property list, separated by ‘;’. No default
SegmentSet	required	replace	RTLoad	Specifies the set of segments that make up a RTLoad. Syntax is a set of SegmentDefs. No default.
Trigger	optional	replace	RTLoad	Specifies a trigger to be used in a RTLoad. Syntax is TriggerDef. no default.

Table 3. Attributes that apply to Players.

3.1.1.2. Implementation of Primitive Components

Primitive components are implemented directly in the code of some programming language or data stored in files. They are made available to UniCon by providing an appropriate specification as a wrapper. Code may be represented as source, object, or executable; other representations will be added as required.

Since multiple representations for a component may be available (for example, both source and object code), UniCon allows multiple representations, called *variants*, to be specified. When a primitive component is instantiated, the Variant attribute can be used to select the preferred variant. This capability can be exploited, for example, to provide variants with different performance properties or variants with special support for debugging or monitoring. Table 4 shows the attributes defined for primitive implementations of components.

The KWIC indexer example is constructed from primitive filter components, like the Sort component shown in Program 1. The interfaces of the component filters are very similar to the interface for the whole system. The implementations are primitive, and since they correspond directly to executable Unix filters, they have only one variant.

Attribute	*Req/Opt*	*Merge Rule*	*Applies to Components*	*Value / Default / Syntax*
ImplType	optional	error	all	Representation: source, object, executable, data, or whatever. Default is to refer to file extension. Syntax is literal from enumeration of known kinds
InitActuals	optional	error	all	Actual parameters for initialization
BuildOption	optional	merge	all	Options passed to the system builder. Syntax is option syntax for system builder. No default.

Table 4. Attributes defined for primitive components.

3.1.1.3. Implementation of Composite Components

Composite components provide the capability of building up progressively larger subsystems from components of (potentially different) types. A composite implementation must provide three kinds of information:

- *The parts:* Instantiations of the components and connectors from which the composite component is constructed.
- *The configuration:* Specification of how the instantiations of connectors link the instantiations of components, i.e. the associations[5] between players and roles.
- *The abstraction:* Specification of how the players of the interface will be associated with players of the implementation.

Since more than one instance of an element (component or connector) definition may be used in a single implementation, these definitions are templates to be instantiated for each use. Each instantiation may provide a property list that further constrains the attributes of the element. The *merge rule* of the attribute determines the interpretation when multiple values are provided for an attribute.

The configuration of the composite implementation is defined by explicitly connecting players and roles. Each match is checked by comparing the type of the player against the Accepts attribute of the component, ensuring that maximum and minimum connection counts are satisfied.

The *interface* of the component being defined specifies the players that the component must provide; these are the *ExternalPlayers*. In the implementation, players are provided when components are instantiated; these are the *InternalPlayers*. The abstraction step defines *ExternalPlayers* in terms of *InternalPlayers*. Two cases arise:

- In the simple case, one of the constituent components defines an InternalPlayer of the same type and specification as the ExternalPlayer. In this case, simple name binding suffices to export the player through the interface.
- The more complex case arises when an ExternalPlayer is of a type not directly supported by the programming language. In this case the ExternalPlayer must be implemented by more concrete InternalPlayers of the implementation. This is the case, for example, when a StreamIn is implemented with calls on library routines that read standard input. It will become increasingly common as we add more abstract players that are realized as calls on several specific procedures according to set protocols. The definition of such an abstract

[5]An association is either a bind (to an external player of a component) or a connect (to a role of a connector).

connector must specify the functionality that the implementation must provide. In this case, the binding must indicate the name of the ExternalPlayer in the interface and show how InternalPlayers satisfy the required functionality. The special attribute MapsTo identifies the InternalPlayer(s).

Warnings about ExternalPlayers and roles that are not associated with InternalPlayers are given as appropriate. Setting the MinAssocs attribute of a Player to 0 indicates that it is normal for the Player to be unassociated. A *Library* attribute will soon be provided to indicate that many players will normally remain unassociated.

In the implementation of KWIC in Program 1, four filters and a file are used to build a system that is itself a filter. The definition of pipe used here corresponds to unix: as indicated in Table 5, it supports the players of sequential files as well as filters. The implementation is composite and has three parts. First, it instantiates the parts to be used. Next, it binds the StreamIn of the first filter (caps) to the StreamIn of the interface and the StreamOut of the last filter (sorter) to the StreamOut of the interface. (This is an example of the simple case where ExternalPlayers are bound to InternalPlayers of the same type.) Finally, it configures the system by indicating which inputs and outputs are connected by which pipes. The example shows two ways to do this: by individually associating players to roles of explicitly instantiated pipe connectors and by a single statement that implicitly instantiates the pipe and makes all connections at once. The former allows the connections to be grouped by the designer’s choice; the latter assures the reader that no other roles of the connector are connected elsewhere. The mechanical character of the example provides ample motivation for using the graphical interface described in Section 3.2.

3.1.2. Connectors

Connectors mediate interactions among components. A connector consists of a protocol that specifies the class of interactions the connector provides and an implementation. Connectors define the protocols and mechanics of interaction together with any additional mechanisms required to carry out the interaction: auxiliary data structures, initialization routines, and so on. The connector definition is also the location for specifications of required behavior such as interchange representations and the internal manifestation (e.g. sequence of procedure calls) of the connector in the code of a component. At present, all connectors are primitive and their implementations are therefore individually crafted within the UniCon implementation.

The protocol defines the allowable interactions among a collection of components and provides guarantees about those interactions. To do this it defines roles, or the responsibilities of various parties that set requirements for the players of components whose interactions are to be governed by the connector. The author of the component is responsible for ensuring that these responsibilities are satisfied by the implementation. The protocol must include:

- the connector type
- assertions that constrain the entire connector (for example, rules about timing or ordering); these are the commitments about the interaction that the protocol supports
- the roles that participate in the protocol; each consists of a name and type and optional attributes like signature, functional specifications, or constraints on their use.

A connector type expresses the designer's intention about the general class of connection to be provided by the connector; it restricts the numbers, types, and specifications of the Roles provided by the connector. In particular, some roles may require players, some may be optional but constrained if present, and some may be restricted to match certain player types. A good example from a programming language would be the generator as defined in Alphard [Shaw 81].

The roles are the visible semantic units through which the connector mediates the interactions among components. Their types are primitive typing units. They are used to identify the players that must cooperate in a successful interaction. Roles identify the kinds of interactions a connector can establish—the kinds of components it can work with and the player types it can handle. When a role appears in a protocol, it must specify a name and role type and may optionally specify other attributes; some of these attributes may be required in particular instances. The detailed specifications of the roles appear in the form of property lists, or lists of attributes and their associated values. Roles form the bulk of the protocol.

```
CONNECTOR Unix-pipe
   PROTOCOL IS
      TYPE Pipe
      ROLE source IS source
         MAXCONNS (1)
         END source
      ROLE sink IS sink
         MAXCONNS (1)
         END sink
   END PROTOCOL

   IMPLEMENTATION IS
      BUILTIN
   END IMPLEMENTATION
END Unix-pipe
```

Program 2. A primitive connector.

At present, only primitive implementations of connectors are supported. Program 2 gives the textual definition of the pipe connector for the KWIC example. These connectors only have primitive bodies, so there is no issue of matching. The bodies are at present defined in an ad hoc manner. Primitive connectors are not further interpreted at the architecture level except through their protocols. This might be as simple as a rule that procedure calls must match procedures in the fashion allowed by the programming language or as complex as a network protocol supported by several independent communication servers.

3.1.2.1. Built-in Connector Types

Connector types and role types are now defined by enumeration, but extensions may someday be supported. Table 5 lists the connector types currently supported, the roles allowed for each, and the players that the roles can support. As for component types, we chose them opportunistically. For connectors, especially, we wanted to deal with the practical details of actual implementations.

Some of these are more closely related than others (the same is true for components). FileIO and Pipe are abstractions over the same Unix mechanisms. The language would benefit from type taxonomies that show these relations. In this case, for example, it would not be necessary to identify all three component types for which GlobalDataUse may be a player. Moreover, General could be treated as the root of a taxonomy rather than as a special case.

Connector type *Pipe* provides the Unix abstraction of pipe. Depending on whether it establishes an interaction between pipes or files, it chooses the correct implementation mechanism. When a system is constructed of many pipes, filters, and files, UniCon creates an initialization routine that starts up the filters with all the proper port bindings; it handles arbitrary topologies correctly. The KWIC example of Figure 1 is based on pipes. Specifically, it uses Unix-pipes, which are of type Pipe. These particular pipes are restricted to a single association, so an attribute is provided to override the default. The protocol specifies two roles, corresponding to the two ends of the pipe.

ConnectorType	*Role Types and the Players they support*
Pipe	Source (accepts StreamOut of Filter, ReadNext of SeqFile) Sink (accepts StreamIn of Filter, WriteNext of SeqFile)
FileIO	Reader (accepts ReadFile of Module) Readee (accepts ReadNext of SeqFile) Writer (accepts WriteFile of Module) Writee (accepts WriteNext of SeqFile)
ProcedureCall	Definer (accepts RoutineDef of Computation or Module) Caller (accepts RoutineCall of Computation or Module)
DataAccess	Definer (accepts GlobalDataDef of SharedData or Module) User (accepts GlobalDataUse of SharedData, Computation, or Module)
PLBundler	Participant (accepts PLBundle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef of Computation, Module or SharedData)
RemoteProcCall	Definer (accepts RPCDef of Process or SchedProcess) Caller (accepts RPCCall of Process or SchedProcess)
RTScheduler	Load (accepts RTLoad of SchedProcess)

Table 5. Built-in connector types and their roles.

Connector type *FileIO* sets up sequential file reading and writing.

Connector types *ProcedureCall* and *DataAccess* provide the architectural abstractions that correspond the usual inter-module connections supported by programming languages. In addition to making these connectors visible at the architecture level, they do type checking of the defines/uses relation on the basis of signatures rather than the spelling of identifier names.

Connector type *PLBundler* supports the abstraction for connecting a collection of procedure or data definitions with their calls or uses. It abstracts from ProcedureCall and DataAccess in the same way that PLBundle abstracts from the corresponding player definitions.

Connector type *RemoteProcCall* corresponds to the RPC facility supplied by the operating system. It relieves the user of the need to work explicitly with complex libraries and generator processes.

Connector type *RTScheduler* mediates competition for processor resources among a set of real-time processes. It requires an operating system with appropriate real-time capabilities. When rate-monotonic scheduling is selected, UniCon invokes a schedulability analysis to check the real-time correctness of the set of processes to be scheduled. This facility is described in Section 3.5

Further specification of each connector is achieved by providing values for certain attributes. These attribute-value associations are made by property lists. Each attribute is relevant to one or more connector types. It may be required or optional. It must specify what to do if multiple values are specified for the attribute, for example in different property lists. The possibilities are currently to use the most recent (replace), to construct a list of all values (merge), and to refuse definitions after the first (error).

Certain attributes pertain to the connector as a whole. For example, a design decision such as which type of Unix pipe or which real-time scheduling algorithm to use applies to the entire connector. Table 6 shows the attributes of entire connectors. Table 7 lists the attributes defined for Roles. The attributes are largely specific to the type of the Role.

Attribute	*Req/Opt*	*Merge Rule*	*Applies to Connectors*	*Value / Syntax / Default*
InstFormals	optional	merge	all	Formal parameter list for instantiation of component. Default is no instantiation parameters. Syntax depends on rules of implementation language.
PipeType	optional	replace	Pipe	Selects which kind of Unix pipe to use. Default is named. Syntax is select from Named, Unnamed.
Match	optional	merge	PLBundler	Identifies connections to be made between members of bundles. Syntax is set of <player, player> pairs. Default is name matching among the participating PLBundles
Algorithm	required	replace	RTScheduler	Names real-time scheduling algorithm to use from enumeration. Default is RateMonotonic.
Processor	optional	replace	RTScheduler	Names processor to run the set of processes on; processor must be running a RT operating system. Default is local.
Trace	required if sched is RMA	error	RTScheduler	Defines a trace of paths through the real-time code. Syntax is RTLoad.Trigger, RTLoad.Segment, RTLoad.Segment, ..

Table 6. Attributes for connectors.

Attribute	*Req/Opt*	*Merge Rule*	*Applies to Roles*	*Value / Syntax / Default*
Accept	optional	merge	all	Identifies types of players that can serve in this role. Default is enumerated in table above. Syntax is (ComponentType.Player type ,) +
MaxConns	optional	replace	all	Maximum number of players the role can be bound to. Default is unlimited. Syntax is integer.
MinConns	optional	replace	all	Minimum number of players the role must be bound to. Default is 1. Syntax is integer.

Table 7. Attributes for roles.

3.1.2.2. Implementation of Primitive Connectors

The implementation details about these connectors are provided in Section 4. Unlike components, the connectors do not correspond to discrete items to be linked in or referenced. In the current system, ProcedureCall and DataAccess connectors use the mechanisms (e.g., the linker) of the underlying language and leave no residue at execution time; if renaming is done by a composition, it is handled with compile-time macros.

Pipe and FileIO connectors use Unix ports, pipes, and files; an initialization procedure sets up the appropriate topology and starts the processes.

For the RemoteProcCall connector, UniCon automatically generates, compiles, and links the "glue code" for the processes doing the RPCs. This glue code collects the arguments of the remote procedures and converts them to messages that are passed between processes.

For the RTScheduler connector, UniCon collects the Trigger and Segment specifications from the RTLoad players and passes them to the Rate Monotonic analysis tool for analysis of the trace information. To establish the connection, UniCon adds Real-Time Mach scheduling information to the processes to make them schedulable in the operating system and makes them available on the target machine for initialization.

For all processes running under Mach or Real-Time Mach, UniCon turns the code specified in the implementations of the Process or SchedProcess component into heavyweight processes. This is not required for Unix processes since they are already heavyweight processes.

3.2. Graphical Notation

In practice, designers rely heavily on diagrams for describing system architectures. Therefore a graphical form of the notation is essential. The graphical notation for UniCon and its user interface allow specifications to be built incrementally. Figures 1, 2, and 3 were produced with the graphical interface, except that the annotations were added manually.

Our emphasis is on defining composite components. Each component corresponds to a window in which the parts, configuration, and abstraction of a composite component are laid out. The frame of a component's definition window is shaded to suggest the type of the component that is being implemented. The designer instantiates components and connectors from a menu of defined types and positions them. Smaller icons on the edges of the components and ends of the connectors represent players. More identifying information is supplied dynamically as the designer's focus moves from one element to another. Each element has an associated detailed definition that can be opened and manipulated during design.

As far as is possible, different types of components, players, and connectors are distinguished iconically. The graphical editor invokes the checking facilities of the language processor to check as much as possible while the diagram is being developed, so errors are largely prevented rather than corrected after the fact. Figure 2-b contains two features of particular note: The clouds on the edges of the *rev* component indicate that an abstraction binding is being used, and the chain links on the connectors between *rev* and the two supporting components show use of definition bundles rather than explicit connections of individual procedures.

The tool does take an initial step toward resolving type mismatches as discussed in Section 2.3. When a connection of RoutineCall and RoutineDef with mismatched signatures is proposed, the editor selects a connector of type TranslatingProcedureCall. Though not yet fully implemented, this mechanism will allow the connection and provide the designer with a code template that translates the calling signature to the declared signature. The designer must fill in the code to correctly perform the translation. When the graphical composition is complete, the tool generates a correct and complete textual representation.

3.3. Textual Notation

UniCon also supports a conventional textual form. Since the language is still fluid, we have chosen a very simple syntax that relies on property lists to provide information. More elaborate concrete syntax will be selected when we relax some of the restrictions noted at the beginning of Section 3 and move on to a successor language. The collected syntax appears in Appendix A.

The textual notation is interconvertible with the graphical notation. On an initial conversion from text to graphics the screen position attributes will be missing, so manual positioning will be required. Any conversion from graphics to text will yield property lists that include screen position attributes; these are parsed but ignored by the tool.

The syntax is illustrated with the textual UniCon definition of the system sketched in Figure 2. At the top level, this system is a pipe-and-filter system. The example shows how one of the filters is implemented in terms of simple procedures and data. We also use the KWIC indexer of Figure 1 as an example. It has five primitive components and one composite component. It uses two types of connectors, for pipes and procedure/data bundles. The stack component and the PLBundler connector are shown in Program 3.

3.3.1. Major Constructs

Section 3.1 presented an overview of the language. It introduced the two fundamental complementary constructs, the component and the connector, and it established the need for separate specification parts and implementations. In the suggestive syntax given here,

- CAPITAL_LETTERS are used for the built-in words of the language;
- ***BoldItalicLetters*** are used for nonterminal definitions;
- *ItalicLetters* are used for fixed enumerations; and
- CapitalsWithLowerCase are used for identifiers supplied by the programmer (often constrained to be of a specific kind).

To avoid tedious syntax with extraneous nonterminals, the expression (*Foo* ;)* denotes a parenthesized list of zero or more *Foo*s separated by semicolons; a + instead of * denotes a list of at least one; a # instead of * denotes zero or one (optional feature). As a special case, { *Foo* } * denotes a sequence of *Foo*s with no punctuation. The syntax for components and connectors is

```
Component         ::=  COMPONENT  CompTemplateName
                       INTERFACE IS
                            TYPE      ComponentType
                            {    PropertyList } #
                            {    PlayerList   } #
                       END INTERFACE
                       IMPLEMENTATION IS
                            {    PropertyList } #
                            PrimComponent  |     CompComponent
                       END IMPLEMENTATION
                       END CompTemplateName
PlayerList        ::=  {    PLAYER  PlayerName     IS    PlayerType
                       {    PropertyList
                            END        PlayerName    }  #  } +
Connector        ::=  CONNECTOR  ConnTemplateName
                      PROTOCOL IS
                          TYPE      ConnectorType
                          {    PropertyList  }  #
                          RoleList
                      END PROTOCOL
                      IMPLEMENTATION IS
                          PrimConnector
                      END IMPLEMENTATION
                      END ConnTemplateName

RoleList         ::=  {   ROLE      RoleName       IS    RoleType
                      {   PropertyList
                          END       RoleName       }  #   } +
```

3.3.2. Primitive Components

Primitive components are introduced by creating interfaces as “wrappers” for code written in a conventional programming language. The implementation part serves to identify the code and provide startup parameters. The actual parameters may refer to the formal parameters of the interface. The system supports multiple versions of implementations. The primary current use is for providing source, object, and executable representations. It will also support different implementation variants for debugging, performance monitoring, different libraries, and different performance characteristics. When a component is instantiated, it may supply an attribute to select a variant.

```
PrimComponent    ::=  {    Implementation    } +

Implementation   ::=  VARIANT    ImplName       IN    FileSpec
                      {    PropertyList
                           END       ImplName       } #    } +
```

```
COMPONENT stack
  INTERFACE IS
     TYPE Computation
     PLAYER stackness IS PLBundle
        MEMBER (init_stack; RoutineDef;
           SIGNATURE (; "void"))
        MEMBER (stack_is_empty; RoutineDef;
           SIGNATURE (; "int"))
        MEMBER (push; RoutineDef;
           SIGNATURE ("char *"; "void"))
        MEMBER (pop; RoutineDef;
           SIGNATURE ("char * *"; "void"))
        END stackness
  END INTERFACE

  IMPLEMENTATION IS
     VARIANT stack IN "stack.c"
        IMPLTYPE (Source)
        END stack
  END IMPLEMENTATION
END stack
```

```
CONNECTOR C-PLBundler
  PROTOCOL IS
     TYPE PLBundler
     ROLE participant IS participant
  END PROTOCOL

  IMPLEMENTATION IS
     BUILTIN
  END IMPLEMENTATION
END C-PLBundler

  IMPLEMENTATION IS
     BUILTIN
  END IMPLEMENTATION
END C-proc-call
```

Program 3. Stack component and procedure call connector.

A typical primitive implementation is the stack component of Program 3. Note that the full set of operations is exported as a single player; this abstraction supports the intuition that the operations form a coherent collection and will usually be used as a whole. Only one implementation is provided, the source code. Contrast this with the definition of sort in Program 1, which provides an executable representation and the switches to provide when the process is started.

3.3.3. Composite Components

Composite components provide the mechanism for building up subsystems from primitive components (compilation units) or smaller subsystems. A composition requires

- instantiations of some components, defined in the ***CompUses***.
- instantiations of any required connectors, defined in the ***ConnUses***.
- association of the Players of some of these components with the Players of the component being defined, established in the ***Bind***
- definition of the interactions among the components, specifically by defining the fashion in which they are connected and associating the ***Players*** of the constituent ***Interfaces*** with the ***Roles*** of the ***Protocols***, enumerated in the ***Connect***

The basic syntax addresses these three points. Because establishing connections in some of the simplest cases can sometimes become tedious, some syntactic sugaring is also provided. The ***PropertyList*** in the USES clause is intended to select properties specific to the instance being created (for example, initialization parameters).

```
CompComponent   ::=  {  CompUses  |  ConnUses  |  Bind  |  Connect  } +

CompUses        ::=  USES   CompInstance   INTERFACE   CompTemplateName
                     {   PropertyList
                         END     CompInstance   } #

ConnUses        ::=  USES   ConnInstance   PROTOCOL    ConnTemplateName
                     {   PropertyList
                         END     ConnInstance   } #

Bind            ::=  SimpleBind   |   AbstractBind

SimpleBind      ::=  BIND   ExternalPlayer   TO   InternalPlayer

AbstractBind    ::=  BIND   ExternalPlayer TO ABSTRACTION
                         PropertyList
                         END     ExternalPlayer

Connect         ::=  CONNECT    PlayerName    TO   RoleName     |
                     CONNECT    RoleName      TO   PlayerName
```

In implementations of components, USES clauses instantiate components or connectors. Properties may be associated with the entire component or with individual instantiations. Attributes defined for the individual instantiations provide values that are combined with values of the template definition according to the merge rule defined for the attribute. The AbstractBind partially supports the abstraction mapping described in Section 2.2; UniCon supports only the simple form in which a player of one type is “blessed” as adequately implementing a player of some other type. A common case is the implementation of StreamOut with printf, as in reverse-filter of Program 4.

The syntax allows the individual connections between players and roles to appear in any order. Often, however, it is helpful to group all the CONNECT statements for one PROTOCOL. In this case the connector instance name is purely local to that group. Syntactic sugar is provided for this case. It allows implicit creation of the connector instance and assurance that all of the associations between ROLES and PLAYERS are established at one time. It is syntactically transformed to the obvious ***ConnUses*** and set of ***Connects***. To allow this, a ***ConnUses*** and its complete associated set of ***Connect***s can be replaced by:

```
Establish        ::=  ESTABLISH    ConnTemplate   WITH
                      {    PlayerName     AS    RoleName       } +
                      {    PropertyList  } #
                           END        ConnTemplate
```

In the second example, the reverser is a filter whose implementation is shown in Program 4. The upper-level system composition, with pipes and filters, is essentially similar to the system in the KWIC indexer example. The reverser filter is a reverse-filter, whose implementation is given here. The interface is the standard filter interface, except that since the code never uses stderr attribute MinAssocs is set to 0 to show that its use is optional. The implementation instantiates three components and connects the procedures and data in the PLBundles that the three components define.

Three parts of the definition deserve special note. First, the main program, reverse, uses a stack but calls the operations by different names from the stack implementation at hand. UniCon's connection rules support this renaming easily. Second, the connection is established between players that abstract stackness from the collection of individual stack operations. Third, the component defines a filter in terms of procedures. This requires an abstraction step. The components must correctly implement streams, and the definition must show the correspondence. The rules for implementing streams are the same in UniCon as in the underlying system; like other aspects of connection definitions, they are currently built in as special cases. The correspondence is shown in the BIND statements. As described in Section 3.1.1.3, the external player type and the internal player type are different. The appearance of the keyword ABSTRACTION indicates that an abstraction mapping rather than a simple renaming is taking place.

```
COMPONENT reverse-filter
  INTERFACE IS
    TYPE Filter
    PLAYER input IS StreamIn
      SIGNATURE ("line")
      PORTBINDING (stdin)
      END input
    PLAYER output IS StreamOut
      SIGNATURE ("line")
      PORTBINDING (stdout)
      END output
    PLAYER error IS StreamOut
      SIGNATURE ("line")
      PORTBINDING (stderr)
      MINASSOCS (0)
      END error
    END INTERFACE


  IMPLEMENTATION IS
    USES rev INTERFACE reverse
    USES stk INTERFACE stack
    USES lib INTERFACE libc

    BIND input TO ABSTRACTION
      MAPSTO (rev.libc.fgets)
      END input
    BIND output TO ABSTRACTION
      MAPSTO (rev.libc.fprintf)
      END output
```

```
/* Connections related to use of stack.
  Note renaming. Checks are based on specs
  (here signatures) rather than names */

    ESTABLISH  C-PLBundler WITH
      rev.stackness AS participant
      stk.stackness AS participant
      MATCH ((rev.stackness.new,
               stk.stackness.init_stack),
           (rev.stackness.no_more,
               stk.stackness.stack_is_empty),
           (rev.stackness.stash,
               stk.stackness.push),
           (rev.stackness.deliver,
               stk.stackness.pop))
      END  C-PLBundler

  /* Connections that set up library calls */
    ESTABLISH  C-PLBundler WITH
      rev.libc AS participant
      lib.libc AS participant
      MATCH ((rev.libc.fgets,
               lib.libc.fgets),
           (rev.libc.fprintf,
               lib.libc.fprintf),
           (rev.libc.malloc,
               lib.libc.malloc),
           (rev.libc.strcpy,
               lib.libc.strcpy),
           (rev.libc.strlen,
               lib.libc.strlen))
      END  C-PLBundler

  END IMPLEMENTATION
END reverse-filter
```

Program 4. Reverse-filter: a non-primitive component with abstraction.

Additional syntactic sugar is provided for the very common case in which (a) procedure calls and data access are being connected, (b) the Definers and Callers use the same names, and (c) there are no duplicate name conflicts. In this case, the connections can be defaulted, thereby avoiding the creation of very long, error-prone lists of the form

```
ESTABLISH ProcedureCall WITH
    Here.F AS Caller
    There.F AS Caller
    Everywhere.F AS Caller
    Etc.F AS Caller
    END ProcedureCall
```

or, even worse, the fully un-sugared form. Similarly, members of PLBundles will be connected by default when it is safe and unambiguous to do so.

3.3.4. Primitive Connectors

UniCon currently supports only built-in, or primitive connectors. We are extending the list of built-ins manually. Our emphasis is on incorporating connectors that are already rich, well worked-out, and supported on the platform of our implementation. Since one of the objectives of this implementation is to gain experience with a rich variety of connectors, we are not yet in a position to define composite connectors. The syntax is similar to that for primitive components.

PrimConnector ::= BUILTIN

3.3.5. Property Lists

Property lists appear frequently. Each starts with an attribute name and contains a parenthesized *Property*.

```
PropertyList      ::=  {    AttributeName  (    Property     )    } +
```

A ***Property*** is any expression that can be properly delimited by the syntax. This is one of the major mechanisms in support of an open system; properties can be collected syntactically and without interpretation, then shipped off to applications capable of analyzing them.

When an attribute is known to UniCon, especially when it is used in UniCon processing, the attribute is parsed and interpreted. However, when an attribute is provided purely in support of an external tool, it may be simply passed to the tool as a completely uninterpreted string. This capability is required to support tools not specifically known to the system.

3.4. Populating the Space of Elements

In order to get meaningful experience with an architecture tool, we must populate its environment with definitions that do useful things. It is far better to populate this space with real elements than to build toy examples with code created for the purpose.

We have chosen connectors of practical utility and built them into the system. Our major reason was to supply a diverse collection of convincing mechanisms. We did that by selecting mechanisms for which someone else had designed the specification (i.e., protocol) and developed the implementation. A second reason for this strategy was to gain enough experience to know what the language must provide to support composite connectors.

Similarly, we chose to accept existing code as primitive components. We do this by creating wrappers that give the component specification as a UniCon interface and refer to one of the native representations of the code (source, object, executable, ...) in the primitive implementation.

To simplify the creation of these wrappers, we built a semiautomatic tool that extracts the externally visible names of a C module and creates a draft UniCon specification. We have not attempted to perform complete type inference for signatures. Rather, the tool derives a good approximation and provides unresolved signatures to the human user for resolution. In practice, the tool does most of the work. Because of the vagaries of C, some names often appear to be externally visible when that was not the programmers intention, so manual checking is required in any case. Using this tool, we have created wrappers for the code of the UniCon processor itself, and it has successfully constructed itself.

3.5. Incorporating Analysis Tools

UniCon can incorporate externally-developed analysis tools. These tools require information in formats that are not in general known to UniCon. We do this by capturing the tool-specific information as (potentially uninterpreted) entries in property lists. UniCon collects the information required by the tool, forwards it to the tool for analysis, and receives the result for appropriate processing (for example, generation of error messages or provision of an attribute value that will affect system configuration).

We currently support one analysis tool. The RMA tool, described in Section 2.4 determines whether a set of real-time processes will be schedulable under a rate-monotonic discipline. The operating system supported is Real-Time Mach [Kitayama et al 93, Nakajima et al 93].[6] We describe both the UniCon facilities and the connection to the analysis tool here.

Figure 3 shows a real-time client-server example, with two processes contending for processor time. The real-time characteristics and requirements of this client-server system are recorded as part of the architectural description, then automatically collected and passed on to a separate analysis tool. This tool, also developed at Carnegie Mellon, performs real-time schedulability analyses based on tabular data in the “Implementation Table” format introduced in [Klein et al 93]. These tables describe the events, actions, and resources of the system. (In this formulation, an event is an abstraction that consists of a stimulus and a set of responses that follow from it.) Tables 1, 2, and 3 comprise the Implementation Table for the example of Figure 3. UniCon allows this information to be recorded in the interfaces of SchedProcess components (see Program 5) and extracts the tables as shown for transmission to the analysis tool.

The RTM-realtime-sched connector establishes a real-time scheduling relation among a number of processes. Since an abstract real-time task may require code executing in several processes, sophisticated scheduling is required. Program 5 presents the significant parts of the code for the example of Figure 3. This includes the component that does the complete system configuration (Real_Time_System), the specification (only) of one schedulable process (RTClient), and the two connectors. The component definitions show how property lists are used to specify the information required by Tables 8, 9, and 10, providing each type of information at the appropriate point in the design. UniCon extracts the information, creates the tables in the appropriate format, and passes them to the RMA analysis tool. The information is also used to initialize the schedulable processes on the correct Real-Time Mach processor under the proper scheduling algorithm.

EVENTS					
Event Name	*Type*	*Mode Name*	*Arrival Pattern*	*Dead-line*	*Responses*
client.application1.external_interrupt1	T	n/a	1000	1000	client.application1.work_block1, server.services.work_block1, client.application1.work_block2, server.services.work_block2, client.application1.work_block3
client.application2.external_interrupt2	T	n/a	500	500	client.application2.work_block1, server.services.work_block1, client.application2.work_block2, server.services.work_block2, client.application2.work_block3

Table 8. Event table for Real-time analysis.

[6]Mach is a microkernel developed at CMU, and a Unix server running on Mach provides a Unix interface and programming environment. RT Mach, also being developed at CMU, is upward compatible with Mach but provides extensions including periodic real-time threads, exception handlers for timing violations, high resolution clocks/timers, real-time synchronization, and real-time interprocess communication. UniCon supports both Mach and RTMach interprocess communication (IPC) facilities.

ACTIONS						
Action ID	*Jitter*	*Resource ID*	*Atomic*	*User ID*	*Time Used*	*Priority*
client.application1.work_block1	n/a	TESTBED.XX.EDU	N	client	20	10
server.services.work_block1	n/a	TESTBED.XX.EDU	N	server	40	9
client.application1.work_block2	n/a	TESTBED.XX.EDU	N	client	30	10
server.services.work_block2	n/a	TESTBED.XX.EDU	N	server	30	9
client.application1.work_block3	n/a	TESTBED.XX.EDU	N	client	50	10
client.application2.work_block1	n/a	TESTBED.XX.EDU	N	client	20	10
client.application2.work_block2	n/a	TESTBED.XX.EDU	N	client	30	10
client.application2.work_block3	n/a	TESTBED.XX.EDU	N	client	50	10

Table 9. Action table for Real-time analysis.

RESOURCES		
Resource ID	*Type*	*Scheduling Policy*
TESTBED.XX.EDU	CPU	Fixed Priority

Table 10. Resource table for Real-time analysis.

In this example, the processes also interact via remote procedure calls; these interactions are mediated by the RTM-remote-proc-call connector. The full text of the example includes one composite component to define the system, two schedulable process components that convert simple procedures to processes, two modules that provide the application code of the real-time tasks, two connectors, and two libraries.

Component type SchedProcess provides an abstraction for processes that must meet real-time deadlines and that the operating system schedules accordingly. These processes may be periodic or aperiodic. Real-time applications use SchedProcess components to define computations based on multiple processes that execute periodically, concurrently, and in competition for the CPU resource. Interactions among SchedProcess components are mediated by connectors of type RTScheduler. This connector type recognizes an Algorithm attribute to choose from among six scheduling algorithms. If the algorithm rate_monotonic is selected, UniCon invokes the RMA analysis tool to determine whether all of the schedulable processes will meet their deadlines.

At this level of abstraction, the designer has not indicated how the realtime scheduling connector is to be implemented. It might be a single scheduling component that registers processes. However, as described in Section 5.1, the intended implementation involves a combination of lightweight and heavyweight processes, in some cases with auxiliary hidden processes. The built-in semantics of the connector component includes the wrappers as well as the initialization code that sets up the operating system scheduler.

```
COMPONENT Real_Time_System
  INTERFACE IS
    TYPE General
  END INTERFACE

  IMPLEMENTATION IS
    USES client INTERFACE rtclient
      PRIORITY (10)
      ENTRYPOINT (client)
      END client

    USES server INTERFACE rtserver
      PRIORITY (9)
      ENTRYPOINT (server)
      RPCTYPEDEF (new_type; struct; 12)
      RPCTYPESIN ("unicon.h")
      END server

    ESTABLISH RTM-realtime-sched WITH
      client.application1 AS load
      client.application2 AS load
      server.services AS load
      ALGORITHM (rate_monotonic)
      PROCESSOR ("TESTBED.XX.EDU")
      TRACE (client.application1.
                   external_interrupt1;
        client.application1.work_block1;
        server.services.work_block1;
        client.application1.work_block2;
        server.services.work_block2;
        client.application1.work_block3)
      TRACE (client.application2.
                   external_interrupt2;
        client.application2.work_block1;
        server.services.work_block1;
        client.application2.work_block2;
        server.services.work_block2;
        client.application2.work_block3)
      END RTM-realtime-sched

    ESTABLISH RTM-remote-proc-call WITH
      client.timeget AS caller
      server.timeget AS definer
      END RTM-remote-proc-call

    ESTABLISH RTM-remote-proc-call WITH
      client.timeshow AS caller
      server.timeshow AS definer
      END RTM-remote-proc-call
  END IMPLEMENTATION
END Real_Time_System
```

```
COMPONENT RTClient
  INTERFACE IS
    TYPE SchedProcess
    PROCESSOR ("TESTBED.XX.EDU")
    TRIGGERDEF (external_interrupt1; 1.0)
    TRIGGERDEF (external_interrupt2; 0.5)
    SEGMENTDEF (work_block1; 0.02)
    SEGMENTDEF (work_block2; 0.03)
    SEGMENTDEF (work_block3; 0.05)
    PLAYER application1 IS RTLoad
      TRIGGER (external_interrupt1)
      SEGMENTSET (work_block1,
        work_block2, work_block3)
      END application1
    PLAYER application2 IS RTLoad
      TRIGGER (external_interrupt2)
      SEGMENTSET (work_block1,
        work_block2, work_block3)
      END application2
    PLAYER timeget IS RPCCall
      SIGNATURE ("mach_port_t",
        "new_type *"; "kern_return_t")
      END timeget
    PLAYER timeshow IS RPCCall
      SIGNATURE ("mach_port_t";
        "kern_return_t")
      END timeshow
  END INTERFACE


CONNECTOR RTM-realtime-sched
  PROTOCOL IS
    TYPE RTScheduler
  ROLE load IS load
  END PROTOCOL

  IMPLEMENTATION IS
    BUILTIN
  END IMPLEMENTATION
END RTM-realtime-sched


CONNECTOR RTM-remote-proc-call
  PROTOCOL IS
    TYPE RemoteProcCall
    ROLE definer IS definer
    ROLE caller IS caller
  END PROTOCOL

  IMPLEMENTATION IS
    BUILTIN
  END IMPLEMENTATION
END RTM-remote-proc-call
```

Program 5. System definition for real-time scheduling example: system definition and connectors

The RMA analysis tool requires information about execution times, periods, relative priorities, and event traces. A Segment corresponds to a block of code and is defined using the SegmentDef attribute. This attribute defines the Segment's name and execution time. A Trigger corresponds to the cause of the event and is either periodic or aperiodic; it is defined using the TriggerDef attribute. This attribute defines the Trigger's name and period. An RTLoad player uses the Trigger and SegmentDef attributes to define the processor load imposed by the SchedProcess in response to a single event. The relative priorities are usually determined for each instantiation; they are provided by attribute Priority, usually when the SchedProcess is instantiated. Event traces describe the sequence of computations that will take place as a consequence of an event. A trace may include segments from many SchedProcesses; it therefore cannot be specified until the processes are associated with a RTScheduler connector. Traces are therefore attributes of the connector. Each consists of a trigger and the segments whose execution follows from that trigger.

4. *Implementation*

The UniCon implementation currently supports the language described in Section 3. The implementation includes tools that process specifications as described in Sections 3.2 to 3.5: a compiler for the textual form, a graphical interface, a semiautomatic wrapper-generator, and a facility for invoking the RMA analysis tool.

Much of the underlying technology is standard. The compiler, for example, is in most respects conventional. Its primary product is Odinfiles[7] rather than machine code; these invoke the language tools that actually construct the systems. It also generates new source code for initialization purposes. The graphical user interface is built using STk, a Scheme interpreter that bundles Ousterhout's Tk toolkit**.** The wrapper-generator is based on the Gnu C compiler with, of course, much simplified processing. The interface to the RMA tool is a data format for a spreadsheet.

The novel part of the UniCon implementation is the handling of connectors, in particular high-level connectors. In UniCon, the implementations of connectors are all built in, the motivation for which was the need to gain experience with a variety of underlying mechanisms. That experience will put us in a position to design language-level constructs to allow the addition of user-defined connectors. We also want to experiment with a wide variety of mechanisms to determine what modifications to the toolset are required for the addition of new mechanisms. As a reality check, we have chosen mechanisms that are available and useful in practice. Finally, by using mechanisms that are already available in our environment, we are confronted with the nitty-gritty details of real implementations. The current connectors are of four essentially different kinds. These are described in the sections below.

4.1. Procedure Call and Global Data Access

In order to produce running systems, UniCon must support some of the interaction constructs that are native to programming languages. In this capacity, UniCon is similar to a module interconnection language. At present the supported constructs are procedure calls and data naming, but future versions may add other constructs such as exceptions and language-based synchronization (e.g., Ada rendezvous). UniCon defines connector types ProcedureCall and DataAccess, and the implementation supports realizations of these in the C programming language, defined as connectors C-proc-call and C-shared-data.

[7]Odinfiles are similar to makefiles, in that they specify system construction steps for producing program executables. Odinfiles are processed by the odin utility, a make-like utility that computes complete dependency informatin automatically. Odin's scripts are shorter and simpler than make's. Odin gains efficiency by eliminating most of the filesystem status queries required by make, by parallel builds on remote machines, and by sharing from a cache of previously computed derived files. For more informatino on odin, contact Geoff Clemm, geoff@bellcore.com.

For each player in a procedure call or global data connection, the compiler keeps track of the location of the source file or object file that implements the player. When the system has been completely parsed and all connections have been successfully made, UniCon creates an Odinfile which compiles all the source files and links all the object files to produce an executable version of the system. The linker is the tool that actually makes the connections for procedure calls and global data access.

It is not always convenient, appropriate, or possible to use the same procedure name at call site and definition site. To overcome this limitation, UniCon allows renaming, as illustrated by stack in the second example. This is implemented by creating C macros to redefine the differing names to a new, unique name. This implementation strategy, of course, is possible only with "source" files; object-level renaming may be added in the future.

In addition to the ESTABLISH abbreviation that allows implicit instantiation of a connector, UniCon allows procedures and data uses to be connected implicitly by matching names. This provides the usual capability of a compiler and relieves the system builder from having to enumerate all of the connections in a complex system. After explicit connections have been performed, any Players with insufficient associations are checked to see whether they can be connected implicitly. If so, they are connected and a warning is issued.

The number of individual procedure call connections between two components can be very large. At the granularity of the system's architecture, it is not helpful to display them all. Furthermore, procedures tend to aggregate into collections of procedures related to some aspect of an interface. In order to support the abstraction of a group of procedure and data connectors, we introduced the player type PLBundle and the related connector PLBundler. The Participant role of the PLBundler accepts both bundled and unbundled players. The PLBundler can connect players of any number of PLBundles; the matching (including renaming) is governed by the Match attribute. In its simplest form it becomes equivalent to a ProcedureCall or DataAccess connector.

4.2. Unix Pipes and Files

In the Unix environment, the most common connector is the pipe as supported by most shells. The shell version of pipe, which allows files to be sources and sinks, is an abstraction over the system mechanisms. At the system level, file operations, named pipes, and unnamed pipes are separate entities that all happen to be accessed through ports. Like shells, UniCon inspects the components that a pipe connects and selects the appropriate mechanism to implement the pipe abstraction. Unlike shells, UniCon provides the same notation for pipe configuration as for others and handles arbitrary plumbing topologies.

UniCon defines the connector type Pipe. The language supports connection of ports to either files or to ports of other processes. The tools select the correct implementation from among the low-level choices provided by unix. This is independent of programming language. The examples in this paper use a particular form of Pipe connector, the Unix-pipe, whose abstraction is appropriate for this operating system. UniCon goes one step beyond Unix in providing a signature for each stream to specify the syntactic structure expected on the stream. UniCon checks these signatures for consistency.

Most unix users regard connecting a sequence of processes together in a pipeline as a simple task. Indeed, most of the myriad unix shells provide a simple syntactic mechanism for doing so. This mechanism allows a one-way data path, with a source, an arbitrary number of intermediate filters, and a sink. Thus, when most people think of pipes in the Unix environment, the notion of reading from "standard input", writing to "standard output", and printing errors on "standard error" (so that they appear on the terminal instead of passing through the pipeline) is the model that comes to mind—indeed, this has been standard for over 20 years [Ritchie & Thompson 74]. Although this model has proven useful, many other configurations of communicating processes are possible, such as the data merging shown in Figure 1, two-way communications, loops, and so forth. So firmly ingrained is this standard mechanism that most people are unaware that pipes can be used to connect arbitrary file descriptors from an arbitrary collection of processes together.

UniCon, on the other hand, supports abstractions that encourage the use of general pipeline topologies. The PortBinding attribute provides for explicit selection of particular ports, and the freedom from linear text allows the creation of complex topologies.

The low-level details of connecting two processes with a pipe are rather complicated. It requires that the processes have a common ancestor (i.e., parent process), and that parent establish the pipe connections before doing the `exec` on the child processes. UniCon uses this model for connecting processes with pipes. The implementation is complicated by a number of additional issues:

- UniCon must dynamically assess the interconnection pathway and build the interconnection code based on an arbitrary linking of processes.
- Because splitting, merging, and data loops are allowed, UniCon must pay careful attention to parenting.
- Unlike the shell, UniCon allows processes to communicate on file descriptors other than `stdin` and `stdout`. The UniCon description permits the user to specify which file descriptors the process expects to be present, and connects the pipe appropriately.

Experience has shown that although the current textual syntax is somewhat cumbersome (the graphical notation is simple), it is very easy to specify complex interconnections such as that found in the KWIC indexer example.

4.3. Remote Procedure Call

Pipes and filters provide data-flow relations among independent processes. Remote procedure call (RPC) complements that with an interprocess relation based on communication and control flow. RPC has a natural abstraction relation with ordinary intraprocess procedure call, but it requires substantial extra effort on the part of the programmer.

UniCon defines connector type RemoteProcCall. The implementation supports both Mach and RTMach versions of this connector as M-remote-proc-call and RTM-remote-proc-call. They are implemented for Mach 3.0 Interprocess Communication (IPC) Facility and RTMach IPC, respectively.

Mach and RT Mach implement RPCs with message passing. The arguments of a remote call are marshaled into a message and delivered to the process that defines the remote procedure; this process unmarshals the arguments and calls the routine in that process; the return value is handled similarly. The glue code that converts simple procedures to remote procedures is very tedious for software engineers to create, so the designers of Mach and RT Mach provide a tool, called the Mach Interface Generator (MIG), that can produce the glue code from simplified specifications of RPC routine definitions.

For each player in a remote procedure call connection, the compiler keeps track of the location of the source and/or object files that comprise the implementation of the process in which the player is found.[8] UniCon also keeps track of the RPCDef players and RPCCall players found in each process. For each process that exposes RPCDef players in its interface, UniCon builds a MIG specification containing the required definitions for each remote procedure declaration in that process. For RPCCall players, UniCon keeps track of the MIG-generated files it needs to link with the process to access remote procedures in other processes.

When the system has been completely parsed and all connections have been successfully made, UniCon creates an Odinfile with the proper system construction steps. For each process doing RPCs the following steps are required:

- Invoke the Mach interface generator for each MIG specification file constructed for the process. This produces C source code for the RPC glue code.
- Compile the C source for the glue code into object files.
- Compile the C source code that turns the Process or SchedProcess implementation into a heavyweight process (see Section 4.1.4).
- If the implementation of the Process or SchedProcess is C source, compile it as well.
- Link all the object files to form the Mach or Real-Time-Mach executable process.

4.4. Real-Time Scheduling

Pipes and remote procedure calls raise the abstraction level of connection, but do not stretch the concept of connection much. It's easy to think of RPCs in the conventional import/export model, and even though import/export doesn't quite match pipes, pipes are still binary and asymmetrical. In order to investigate connectors of a substantially different form, we incorporated a connector for real-time scheduling of type RTScheduler. The implementation supports the RT Mach realization, defined as connector RTM-realtime-sched. In the simplest case, with independent processes, no data or control passes between processes; they interact through competition for processor resources. As described above, the scheduling task becomes more complex when additional interactions are introduced, as through RPC.

The priority and period information for each SchedProcess component is sufficient for UniCon to schedule the process in RT Mach. If the scheduling policy for the computer has been set to rate_monotonic (with the Algorithm attribute) UniCon also packages up the trace, period, execution time, and priority information, transmits it to the analysis tool for schedulability analysis, and reports the results of the analysis.

Next, UniCon generates code for the schedulable processes. In RT Mach, the basis of scheduling is lightweight processes, called threads.[9] However, our real-time model requires scheduling for heavyweight processes. The C code specified in the implementation of the SchedProcess component

[8]More than one Unix file is associated with a Mach or RT Mach process implementation. See section 4.1.4 for further details.

[9]In Unix, a heavyweight process has only one thread, namely "main". In Mach and RT Mach, heavyweight processes can have more than one thread that execute in parallel. In RT Mach, these threads are schedulable.

is the implementation of the thread to be scheduled. In order to satisfy both the model and the operating system, UniCon builds a heavyweight process that creates, initializes, and schedules the thread to run in the RT Mach environment. The resulting thread will run with the period and priority specified. UniCon then builds a startup program that sets the scheduling policy in the RT Mach environment to the one specified in the Algorithm attribute. It also generates Odinfile instructions that build the SchedProcess component. If the SchedProcess component makes remote procedure calls, the generation of the Odinfile is deferred until all information about necessary glue code files is known (see Section 4.3 above). Finally, UniCon generates a shell script that initializes the RT Mach system that the builder has just created. It executes the startup routine which sets the scheduling policy for the processor and then sets the heavyweight processes in motion.

5. *Experience and Analysis*

The UniCon processor is in many respects a quite conventional compiler. Its novelty lies in the variety of high-level connectors it supports and the system configurations it can describe. The substantial work of the development has not been the creation of the software itself, but simply understanding what's required to achieve the desired functionality. This often involves re-examining standard system facilities (e.g., linkers, pipes) in order to use them in more general ways than is now customary.

We demonstrated the first version of UniCon in 1992. Both the language and the implementation are in their second versions. We can now report on our experience with the system. Section 5.1 discusses our experience with the system and is organized generally around types of connectors. Section 5.2 reports some performance data. Finally, Section 5.3 looks to the future.

5.1. Experience

The examples in this paper are among the simpler working examples from each of the architectural styles we support. This section describes the implementations and some of the issues they brought to light.

For pipes, files, and filters, the major challenge was implementing a general model that permitted arbitrary configurations as described in Section 4.2. In addition to the matter of startup order, there is also a matter of the correct order of construction: Unix processes do not actually have ports until they are initialized at execution time. As a consequence, UniCon must set up code to create pipe linkages at process initialization time.

To verify that UniCon provides a smooth bridge from architecture to code, we built the UniCon compiler from a UniCon specification of itself. The compiler consists of 15 C source code modules, 1 lex specification, 1 yacc specification, and 16 C include files. This yields a total of approximately 11,900 lines of source of all types. The UniCon specification was generated semi-automatically using the wrapper-generator discussed in Section 3.4 and was expressed in terms of the raw procedures and data of the implementation. The result was fine-tuned by hand to produce the final specification. On a Sun SPARCstation 10, building UniCon from source code took 1 min 43 sec to complete (wall clock time). Of this, 1 min 24 sec was spent in the Odin utility compiling and linking the system, leaving 19 sec of time attributable to the UniCon compiler.

Procedure call and pipe connectors cannot be mixed arbitrarily. Procedure calls take place within a single address space, whereas pipe connectors are only meaningful at a process (hence name-space) boundary. As a result, a composite implementation may either establish procedure call and data access connections, or it may establish pipe connections. At present, the designer is responsible for determining where process boundaries will fall. The component type structure helps the designer identify the boundaries.

The usual method for using remote procedure call (RPC) in Mach requires substantial knowledge of the underlying run-time library facilities, and at the very least a knowledge of the Mach Interface Generator (MIG). UniCon saves the system builder from having to acquire any of this specialized knowledge. When the RPCs make use of the simple, standard C language types known to the MIG, the system builder need only connect the players in a UniCon specification to create a RPC connection. This takes 3 or 4 lines for each call (one USES and two CONNECTs, or one ESTABLISH). If complex types are involved, the builder need only add another line for each type definition (the RPCTypeDef attribute) and one line for all the include files where the types can be found (the RPCTypesIn attribute). For Example 3 (Figure 3, Program 5, Appendix D), two RPC calls and one complex type definition required 10 lines of UniCon code. UniCon builds the glue code from these, hiding the details from the system builder completely. If the system builder were not using UniCon, but rather developing the application by hand, the builder would be required to know the MIG specification language, create a MIG specification of the glue code, and then build an Odinfile that invokes the MIG to create the glue code and compile and link it into the appropriate processes. If the system builder were not using the MIG at all, but rather generating the glue code by hand, building the system would require a very specialized knowledge of Mach and approximately 500 lines of C code.

This example also reminded us that real code is often not as pure as the models suggest. The modules with the actual code for the real-time tasks called an assortment of library routines, including printf and a routine to convert kernel return codes to error strings. It turns out that these display information in a window to show the progress of the tasks, which is a surrogate for performing some real-time task. However, our client did not originally include this external connection as part of the interface of the component. This will turn out to be a relatively common case, as hooks for debuggers, performance monitors, audit trails, and other tasks not directly related to the overt function of the system are generally not regarded as part of its interface. We believe that some facility for structuring the full interface of a component into the facets of interest to various classes of users will be required; the PLBundle player type is an initial step.

Similarly, running real-time applications under RT Mach requires considerable overhead in process and thread definition. UniCon makes this process transparent. In the implementation of a SchedProcess component, the system builder specifies the implementation of the RT Mach thread, not the process. UniCon then builds the heavyweight process "main" program that creates and properly schedules the thread. UniCon compiles and links the thread and the main program to form the RT Mach process. The system builder need not know any of the RT Mach underlying mechanism for building processes. Mach 3.0 has the same models for lightweight threads and heavyweight processes as RT Mach, so UniCon uses the same model for (non-real-time) processes in both operating systems.

From the implementation of the SchedProcess component type and the RTScheduler and RemoteProcCall connector types we learned that UniCon is simple enough and "open" enough to support the addition of new component, connector, player, and role types. In addition, the compiler proved to be relatively easy to modify to include the new underlying mechanism for creating RT Mach processes and for facilitating inter-process communication via RPC. Both of these experiences indicate that we should find little difficulty in populating the UniCon language with new component types and connector types and building systems from compositions of them. This should help us close in on our goals of producing a formal taxonomy of component and connector types and of taking advantage of useful mechanisms that already exist.

5.2. Performance

The cost of using UniCon has two components: the extra runtime cost imposed on the systems it produces and the setup before runtime. The relevant cost measures are computation time for both components and human time for setup. Whereas Section 5.1 discusses savings in human time that result from not having to learn and operate sophisticated interaction mechanisms, this section discusses computation time.

UniCon does not impose execution performance penalties after initialization. Some connectors require initialization routines; beyond that, the running code is essentially the same as a programmer would likely have produced without UniCon. More specifically,

- For systems of procedure calls and shared data, the executables produced by UniCon are the same size as those that would be produced by hand by a system builder manually compiling and linking the source files.
- Systems of pipes and filters require an initialization process as described in Section 4.2 to establish the topology and set up the pipes. After this process is finished, no residue of UniCon remains. UniCon initialization is slightly faster than the Unix shell; the same algorithms and mechanisms are used in both cases, but the shell must parse and interpret the commands.
- Similarly, there is no runtime performance penalty for systems of schedulable processes built by UniCon. Again, the executables are the same size as those that would be built by a system builder. This, however, assumes that a hand-built system would adhere to the same stylistic restrictions that UniCon does, specifically the mapping of abstract real-time tasks to a set of single-threaded heavyweight processes under RT Mach.
- For remote procedure calls, UniCon automates the recommended process of using the Mach Interface Generator.

To provide a feeling for pre-runtime costs, Table 11 shows the actual build times for our three examples.

Example	*Total Build Time*	*Time in Odin*	*Time in UniCon*
1: KWIC pipes	13 seconds	11 seconds	2 seconds
2: heterogeneous	30 seconds	24 seconds	6 seconds
3: real-time	6 minute 12 seconds	6 minute 5 seconds	7 seconds

Table 11. Build times for three examples

Additionally, the wrapper-generator discussed in Section 3.4 was used to produce some of the UniCon specifications in the examples. The execution time for the wrapper-generator to produce the components stack and reverse (see Appendix C) was approximately 1 second. The wrapper-generator took between 1 and 16 seconds to produce each wrapper for the UniCon specification of itself.

5.3. Conclusion

We set out to provide improved support for abstractions that software developers use in designing the architectures of software systems. This required models that build up from the current module interconnection tools. We developed notation and tools to implement the new models and (1) demonstrated their adequacy for existing tasks (i.e., nothing serious left out) and (2) argued by example their appropriateness for the currently-unsupported activities designers appear to engage in.

Until now, compositional design has been based primarily on ad-hoc choice, informal experience, and local expertise. This work is a step toward making this knowledge precise, semantically-based, and available to engineers as a matter of routine practice. Just as we now have a science of algorithms and data structures for making design tradeoffs and reasoning about properties of code, we shall also have a science for supporting large-scale composable systems.

The principled use of compositional structures will have a dramatic effect on software production. It will (1) permit choosing design paradigms to match desired system characteristics, (2) allow developing application-specific frameworks and reference architectures, (3) enable development techniques (e.g., formal analysis, software reuse) to exploit compositional properties of systems, (4) support a high level of parameterization so that large systems can be more easily designed, understood, maintained, and enhanced, and (5) enable reusing old code as well as provide ways to recover partial architectural information from existing systems.

Acknowledgments

Our collaborators at Carnegie Mellon have provided a continuing forum for exploring these ideas. We particularly appreciate critical and stimulating discussions with colleagues in the School of Computer Science, especially David Garlan, and with participants in the regular research seminar, especially Judy Bishop. Much of the experience we needed to implement the SchedProcess component in UniCon came from a collaboration with engineers, especially Raj Rajkumar, from Carnegie Mellon's Software Engineering Institute, where they are using UniCon as a front end to describe, analyze, and prototype real-time systems. Fuchun Jiang worked on an early version of the system.

This research was supported by the Carnegie Mellon University School of Computer Science and Software Engineering Institute (which is sponsored by the U.S. Department of Defense), by a grant from Siemens Corporate Research, and 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. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. 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 Department of Defense, the United States Government, Siemens Corporation, or Carnegie Mellon University. Theodore L. Ross was supported through Digital Equipment Corporation's Graduate Engineering Education Program.

Appendix: Collected Syntax

```
Component ::=
   COMPONENT   CompTemplateName
      INTERFACE IS
         TYPE  ComponentType
         {  PropertyList  } #
         {  PlayerList  } #
      END INTERFACE
      IMPLEMENTATION IS
         {  PropertyList  } #
         PrimComponent |
         CompComponent
      END IMPLEMENTATION
      END CompTemplateName

PlayerList ::=
      {  PLAYER  PlayerName  IS  PlayerType
      {  PropertyList
         END   PlayerName  }  #   } +

Connector ::=
   CONNECTOR   ConnTemplateName
      PROTOCOL IS
         TYPE  ConnectorType
         {  PropertyList  } #
         RoleList
      END PROTOCOL
      IMPLEMENTATION IS
         PrimConnector
      END IMPLEMENTATION
      END ConnTemplateName

RoleList  ::=
      {  ROLE  RoleName   IS  RoleType
      {  PropertyList
         END   RoleName   }  #   } +

PrimComponent ::=
      {  Implementation  } +

Implementation ::=
      VARIANT   ImplName   IN   FileSpec
      {  PropertyList
         END   ImplName   } #   } +

CompComponent  ::=
      {  CompUses |  ConnUses  |
   Bind |  Connect   } +

CompUses ::=
      USES  CompInstance
         INTERFACE  CompTemplateName
      {  PropertyList
         END   CompInstance   } #

ConnUses ::=
      USES  ConnInstance
         PROTOCOL  ConnTemplateName
      {  PropertyList
         END   ConnInstance   } #

Bind ::=   SimpleBind  |  AbstractBind

SimpleBind  ::=
      BIND  ExternalPlayer   TO   InternalPlayer

AbstractBind ::=
      BIND  ExternalPlayer TO ABSTRACTION
         PropertyList
         END  ExternalPlayer

Connect  ::=
      CONNECT   PlayerName  TO   RoleName   |
      CONNECT   RoleName    TO   PlayerName

Establish  ::=
      ESTABLISH  ConnTemplate  WITH
      {  PlayerName  AS   RoleName   } +
      {  PropertyList  } #
         END  ConnTemplate

PrimConnector  ::=   BUILTIN

PropertyList ::=
      {  AttributeName   (  Property   )  } +
```

References

[Abowd et al 93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. *Proc First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dec 1993.

[Allen & Garlan 94a] Robert Allen and David Garlan. Beyond Definition/Use: Architectural interconnection. *Proc Workshop on Interface Definition Languages*, 1994.

[Allen & Garlan 94b] Robert Allen and David Garlan. Formalizing Architectural Connection. *Proc Sixteenth International Conference on Software Engineering*, 1994.

[Barstow & Wolf 93] David Barstow and Alex Wolf (track chairs). Design Methods and Software Architectures Track. *Proc 7th International Workshop in Software Specification and Design*, IEEE Press, 1993.

[Bell & Newell 71] C. Gordon Bell and Allen Newell. *Computer Structures: Readings and Examples*. McGraw-Hill 1971.

[Boehm & Scherlis 92] Megaprogramming. *Proc Software Technology Conference 1992*, DARPA.

[Callahan & Purtilo 91] John R. Callahan and James M. Purtilo. "A Packaging System for Heterogeneous Execution Environments." *IEEE Trans. on Software Engineering*, 17(6): 626-635, June 1991.

[Campos & Estrin 78] SARA Aided Design of Software for Concurrent Systems. *Proc. National Computer Conference*, 1978.

[Cooprider 79] L. W. Cooprider. *The Representation of Families of Software Systems*. PhD Thesis, Carnegie Mellon University. Apr 1979.

[Dean & Cordy 93] Thomas R. Dean and James R. Cordy. Software Structure Characterization Using

Connectivity. *Proc. Workshop on Studies of Software Design*. Lecture Notes in Computer Science, Springer-Verlag, to appear 1995.

[DeRemer & Kron 76] Frank DeRemer and Hans H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Trans. on Software Engineering,* SE-2(2):80-86, June 1976.

[Garlan & Shaw 93] David Garlan and Mary Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds), *Advances in Software Engineering and Knowledge Engineering,* vol. 2, World Scientific Publishing Company, 1993, pp.1-39.

[Garlan & Shaw 94] David Garlan and Mary Shaw. Software Development Assignments for a Software Architecture Course. Submitted for publication.

[Habermann & Tichy 92] Nico Habermann and Walter Tichy. *Future Directions in Software Engineering.* Dagstuhl Seminar Report 32, Feb 1992.

[Hayes-Roth & Tracz 93] DSSA Tool Requirements for Key Process Functions. Unpublished manuscript, version 1.0, Oct 1993

[Hoare 72] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica,* vol 1 no 4, 1972 pp. 271-281.

[Kitayama et al 93] Takuro Kitayama, Clifford W. Mercer, Tatsuo Nakajima, Stefan Savage, Hideyuki Tokuda, and Jim Zelenka. *Real-Time Mach 3.0 User Reference Manual.* School of Computer Science, Carnegie Mellon University, Pittsburgh PA Aug 1993. Preliminary edition.

[Klein et al 93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harobur. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

[Lam & Shankar 94] A Theory of Interfaces and Modules I—Composition Theorem. *IEEE Tr on Software Engineering,* 20, 1, Jan 1994, pp.55-71.

[Lamb 95] David A. Lamb (ed). *Proc. Workshop on Studies of Software Design.* Lecture Notes in Computer Science, Springer-Verlag, to appear 1995.

[Magee et al 89] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in CONIC. *IEEE Transactions on Software Engineering*, SE-15 (6) pp.663-675, 1989.

[Magee et al 93] Jeff Magee, Nandakar Dulay, and Jeff Kramer. Structuring parallel and distributed programs. *Software Engineering Journal* 8 (2) pp.73-82, March 1993.

[Mettala & Graham 92] Erik Mettala and Marc H. Graham. *The Domain-Specific Software Architecture Program.* CMU/SEI Report CMU/SEI-92-SR-9, June 1992.

[Mitchell et al 79] J. G. Mitchell, W. Maybury, and R. E. Sweet, *Mesa Language Manual*. Tech. Report CSL-79-3, Xerox Corporation, Palo Alto Research Center, Apr 1979.

[Nakajima et al 93] T. Nakajima, T. Kitayama, H. Arakawa, and H. Tokuda. "Integrated Management of Priority Inversion in RT-Mach." *Proc. IEEE Real-Time Systems Symposium,* Dec 1993.

[Perry 87] Dewayne E. Perry. Software Interconnection Models. *Proc. Ninth International Conference on Software Engineering*, IEEE Computer Society Press, Mar 1987.

[Perry & Wolf 92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40-52, Oct 1992.

[PLoP 94] *Proc. First Annual Workshop on Pattern Languages for Programming.* To appear.

[Prieto-Diaz & Neighbors 86] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4), Nov 1986, pp. 307-334.

[Purtilo 90] James Purtilo. *The Polylith Software Bus*. Dept. of Computer Science, Univ. Maryland, Tech. Rep. 2469, 1990.

[Rapide 93] The PAVG Group. *The Rapide-1 Executable Language Reference Manual; the Rapide-1 Types Reference Manual.* Stanford University reports, Mar 1993.

[Ritchie & Thompson 74] D. M. Ritchie and K. Thompson. "The UNIX Time Sharing System." *Comm ACM*, 17, 7, July 1974, pp. 365-375.

[Shaw 81] Mary Shaw (ed.) Alphard: Form and Content. Springer-Verlag 1981.

[Shaw 88] Mary Shaw. Toward Higher-Level Abstractions for Software Systems. *Proc. Tercer Simposio Internacional del Conocimiento y su Ingerieria*, Oct 1988.

[Shaw 95] Mary Shaw. *Making Choices: A Comparison of Styles for Software Architecture*. Carnegie Mellon University Technical Report, 1995.

[Shaw & Garlan 93] Mary Shaw and David Garlan. *Characteristics of Higher-level Languages for Software Architecture*. Carnegie-Mellon University Technical Report, 1993.

[Thomas 76] J. W. Thomas. *Module Interconnection in Programming Systems Supporting Abstraction*. PhD Thesis, Brown University. June, 1976.

[Tichy 79] Walter F. Tichy. Software Development Control Based on Module Interconnection. *Proc. 4th International Conference on Software Engineering*, Munich, 1979, pp. 29-41.

[Wing 94] Jeannette M. Wing (ed). *Proc. Workshop on Interface Definition Languages.* Carnegie Mellon technical report CMU-CS-94-WIDL-1, Jan 1994.

Footnotes