Abstract

The architecture of a software system shows how the system is realized by a collection of components and the interactions among these components. Conventional design focuses on defining the components, but the properties of the system depend critically on the character of the interactions. Although software designers have good informal abstractions for these interactions, the abstractions are poorly supported by the available languages and tools. As a result, the choice of interaction is often defaulted or implicit rather than deliberate choice; further, interactions may be defined in terms of underlying mechanisms rather than the designers’ natural abstractions. UniCon provides a rich selection of abstractions for the connectors that mediate interactions among components. To create systems using these connector abstractions, you need to produce and integrate not only the object code for components, but also a variety of other run-time products. To extend the set of connectors supported by UniCon, you need to identify and isolate many kinds of information in the compiler, graphical editor, and associated tools. This paper describes the role of connector abstractions in software design, the connector abstractions currently supported by UniCon, and implementation issues associated with supporting an open-ended collection of connectors.

Keywords: software architecture, connectors, software system organization, architectural abstraction, architecture description language, system configuration

1. Introduction

Software developers frequently describe a design as a collection of interacting components. For example, a designer may describe a system as a set of real-time processes interacting via remote procedure calls; or as a set of independent experts interacting through a shared blackboard; or as a dataflow architecture with information flowing via pipes through a set of filters [GS93, PW92]. The designer typically focuses on the components: decomposing system functionality into components, choosing representations, and defining interfaces. The choice of how the components should interact—the connectors—is often made implicitly or by default. Moreover, even when designers think about component interactions in terms of abstract relations, the system description itself usually refers directly to low-level mechanisms for communication, control, or data sharing. Such design relegates architectural connections to second-class status and leads to several problems [Sh93].

First, conventional design methods make it hard to localize information about interactions among components. These methods make it easy to identify individual components, because components are usually manifest in the source code. However, identifying the code fragments that correspond to a connection is much more difficult, as this code is often diffuse, implicit, or mingled with code with different overt functions. More seriously, the designer’s abstract intent may be lost entirely, the code recording only the way the designer realized the abstraction in terms of system calls or other low-level mechanisms.

• For remote procedure calls between real-time processes, details are split between the source code and the inputs to a stub generator. The required sequencing of procedure calls may not be recorded at all.

• In a blackboard system, the opportunistic control-flow interaction between experts and the blackboard is emulated in tables, queues, and procedure calls that, in effect, implement an interpreter for the blackboard abstraction.
Pipe connections in a dataflow system are expressed with shell commands (for simple topologies) or operating system calls (for complex topologies). In both cases the format of the data flowing through the pipes is hidden in the parsing code of the filters.

When information about connections is scattered about this way, the system connectivity is hard to discern and the connection mechanism is hard to reuse. Components may also be difficult to reuse, as they are likely to contain embedded connection information.

A second adverse effect of implicit connector information is that it obscures the intended abstractions about component interactions. Labeled lines between boxes in an architectural diagram represent abstractions about the system, such as protocols or shared representations. If this information is not a permanent, explicit part of the system description, its integrity will suffer during maintenance.

A third problem is that components are packaged in expectation of certain kinds of interactions. In Unix, for example, the filter version of sort is not interchangeable with the system call for sort. When packaging and connection expectations are hidden, systems may unintentionally use components with different and incompatible packaging, resulting in serious integration problems. We believe that this is a significant source of difficulties with software reuse [Sh95].

We are addressing these problems in UniCon, an architectural description language that makes connectors—relations between components—first-class constructs in the language. Section 2 summarizes the UniCon language. Section 3 describes the requirements that first-class connectors impose on the UniCon compiler. It also describes our strategy of progressive codification of connector expertise in the compiler [SG95]. Section 4 gives an extended example of connector expertise. Section 5 summarizes related work.

2. Connector Abstractions in UniCon

2.1. Overview of UniCon

UniCon is an architecture description language organized around two symmetrical constructs: A system is composed from identifiable components of various types that interact via connectors in distinct, identifiable ways. Components are specified by interfaces; they correspond roughly to compilation units of conventional programming languages and other user-level objects (e.g., files). Connectors are specified by protocols; they mediate interactions among components. That is, they define the rules governing component interaction and specify any auxiliary implementation mechanisms required. Connectors do not in general correspond directly to compilation units; they are realized as table entries, linker instructions, dynamic data structures, system calls, initialization parameters, utility servers, and so on. An architectural style is based on selected types of components and connectors, together with rules about other properties of the system, such as connection topology.

A component’s interface consists of the component’s type, specific properties (attributes with values) that specialize the type, and a list of points (players) through which the component can interact with the outside world. Each player is typed and may list properties that further specify the player. A component’s implementation may either be primitive or composite. A primitive implementation consists of some element outside of UniCon’s domain, such as a source file in a given programming language, an object file, a data file, or an executable. A composite implementation consists of other components and connectors, composed as described below.

A connector’s protocol consists of the connector’s type, specific properties that specialize the type, and a list of points (roles) at which the connector can mediate the interaction among components. Each role is typed and optionally lists properties that further specify the role. UniCon currently supports only built-in connectors, so each connector’s implementation is specified as builtin. The experience with the built-in connectors described in this paper will eventually allow us to create constructs for composite and user-defined connectors.

A composite component implementation has three parts. First, uses statements instantiate the parts to be composed. Next, connect statements show how players of the components satisfy roles of the connectors. This creates a configuration of components and the connectors that join them. Finally, bind statements map the external interface to the internal configuration. Given a complete description of the software architecture, the UniCon compiler performs many checks: that a component’s interface or connector’s protocol is consistent with
its type; that a player is connected to a role only when it is capable of fulfilling that role; and that a configuration formed in a composite component fulfills that component’s interface. When the checks are satisfied, UniCon creates the intermediate and final products (see Section 3.1) required to construct and execute the system. The details of the language and its compiler appear in a more complete description of UniCon [SDKRYZ95]. Here we focus on the connector abstractions and their implementation issues.

The UniCon tools allow a software system to be described interchangeably in a graphical and textual notation. Figure 1, a screen from the UniCon graphical editor, shows a real-time system involving a client and server that communicate via RPC and compete for real-time response from the processor. Through interactions with dialog boxes, the editor allows the user to inspect and edit properties that are not depicted. Figure 2 gives a flavor for the textual notation by showing two of the definitions from this system’s description, namely the client component and the real-time scheduler connector. The textual notation is a flatter information representation, where all of an architectural element’s details are manifest at once.

2.2. UniCon’s First-Class Connectors

Using the real time system from Figure 1 as an example, we can see how a UniCon description of a system addresses the concerns raised in Section 1. First, a UniCon connector provides a natural home at which to localize information about a particular interaction among components. The RTScheduler connector, for example mediates competition for a shared CPU resource among process components, such as “client” and “server” in Figure 1. This connector type recognizes an Algorithm property to choose from among six scheduling algorithms. The RTScheduler connector provides a natural home for capturing the details of composing and initializing a real time system on top of a particular real-time operating system, such as Real-

Time Mach. Associated with this connector is the knowledge of how to generate source code to initialize each of the processes at runtime, to generate a program to initialize the scheduler in the OS kernel at runtime, and to generate a shell script to initialize the entire system. Connectors further provide a home for static analyses. If the RTScheduler’s Algorithm is “rate monotonic,” for example, UniCon invokes an external analysis tool to determine whether all of the schedulable processes will meet their deadlines under this scheduling discipline.

Next, UniCon allows the designer’s intended abstractions to be an explicit and permanent part of the system description through types. Every connector is tagged with a type, which is manifest as a type construct in the written notation and as the choice of icon in the graphical notation. This type indicates which roles must be satisfied for the connector to operate properly, together with the types of players that are eligible to play the roles. Allen and Garlan have explored formal specifications of the roles [AG94]. Property lists are used...
to refine the types to subtypes or to specialize a type to a particular use.
component Real_Time_System
    interface is
        type General
    end interface
end component

implementation is
    uses client interface rtclient
        PRIORITY (10)
        ENTRYPOINT (client)
    end client

    uses server interface rtserver
        PRIORITY (9)
        RPCTYPEDEF (new_type; struct; 12)
        RPCTYPESIN ("unicom.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
        IDLTYPE(Mach)
    end RTM-remote-proc-call

    establish RTM-remote-proc-call with
        client.timeshow as caller
        server.timeshow as definer
        IDLTYPE(Mach)
    end RTM-remote-proc-call

end implementation
end Real_Time_System

connector RTM-realtime-sched
    protocol is
        type RTScheduler
        role load is load
    end protocol
end connector

3. Implementation Issues for Connectors

A conventional compiler generates a block of object code for each module of source code. The compilation task for connectors is quite different, however, because most connectors do not correspond directly to discrete units of object code. Connectors are rather translated into a variety of different intermediate and final products that serve to realize the connections in the target system during system construction.

UniCon currently supports several classes of connectors, and the set grows steadily. Data flow connectors support systems in which the computation is paced by availability of data (Pipe). Procedural connectors move the thread of control from one procedure or process to another; they include local and remote procedure calls (ProcedureCall, RemoteProcCall). Data sharing connectors allow data to be exported by one component and imported by another (DataAccess). Resource contention connectors abstract the system support required when interaction takes the form of competition for resources rather than by exchange of information or control (RTScheduler). Aggregate connectors begin introducing abstractions with larger granularity than the discrete underlying mechanisms (PLBundler). All of these are translated to the standard implementations provided by programming languages and operating systems.

We begin in Section 3.1 with the variety of products required to realize a connector. Some of these are used during system analysis, construction, and initialization; others become part of the code in the target system. A major thread of this research involves discovering the knowledge required to produce these products correctly. Section 3.2 discusses our strategy, beginning with exploratory implementation and progressing through successively more rigorous codification. As we have codified the required knowledge, we have been able to localize the expertise required for each type of connector. Section 3.3 describes the kinds of expertise required, and Section 3.4 shows how it is used in the compilation and construction process.
3.1. Realization of Connectors

Each connector type has a concrete, though diffuse, realization in the implementation of a UniCon-defined system. These realizations are of several different kinds, including source code, analyses, component configurations, and directives to system services. To this end, UniCon produces intermediate products used during system construction and initialization as well as final products that persist into execution. Each connector requires a different collection of these intermediate products. The products fall into four major categories (shown with the connectors they support):

- **generated code:**
  - remote procedure call interfaces specified in an intermediate language (RPC)
  - C source code to support process initialization at runtime (RPC, RTScheduler)
  - C source code to perform system initialization at runtime (Pipe)
- **system analysis:**
  - data for real-time schedulability analyzer (RTScheduler)
- **system construction:**
  - Odin\(^1\) system construction instructions (all)
  - macros for renaming, used in Odin scripts during compilation step (RPC, ProcedureCall, DataAccess)
- **system initialization:**
  - Unix shell script (RTScheduler)
  - program executable for environment initialization at runtime (RTScheduler)

The products that persist into execution are essentially the same as those now produced by hand. As a result, UniCon imposes no performance cost after runtime initialization.

\(^1\)Odin is a system construction utility similar to the “make” utility in Unix [C195]. System construction instructions are specified in an Odinfile, similar to a Makefile, which Odin uses to compute complete dependency information 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 information on Odin, contact Geoff Clemm, geoff@cs.colorado.edu.

3.2. Development Strategy

UniCon connectors are evolving in stages, following a strategy of *progressive codification*. We are moving from ad hoc implementations of connector expertise to automatic generation of this expertise from specifications in an abstract notation.

In the first stage, we provided ad hoc support for a diverse set of connectors (Pipe, ProcedureCall, RemoteProcCall, DataAccess, and RTScheduler) in order to understand the implementation implications of first-class connectors. We discovered the kinds of tasks the UniCon compiler must carry out and the kinds of intermediate and final products required for each type of connector. The first prototype achieved these goals and was frozen in May, 1994.

In the second stage, we organized and consolidated our understanding of the knowledge required to handle connector abstractions. Our objective was to identify tasks, types of knowledge, and intermediate products required to implement *every* connector type. We defined *experts*, or collections of connector-specific knowledge required for compilation and construction. Each connector expert contains the knowledge required to implement connectors of that type, including:

- rules, icons, literals for enumerations, table entries, and source code fragments for checking the syntax and semantics of the type
- source code fragments for building the attributed syntax tree
- source code fragments for performing analyses
- source code fragments for making automatic connections
- generators and templates for products used in construction and analysis of the target system; instructions for building the target system; and knowledge of how to initialize connectors of the type in the target system at runtime

In the second stage, we reorganized the first prototype’s implementation to localize commonalities in this knowledge, making it easier to add new connectors and to advance to the later stages. This compiler implementation is in most respects conventional, with the usual lexical, syntax, and semantic analysis phases, a tree-building phase, and a “code generation” phase.
where more than just source code is generated (see Section 3.1). This prototype also has a system analysis phase in which portions of the design are analyzed, and a complex system construction phase that involves many kinds of intermediate products. The second prototype was frozen in October, 1994. Two new connectors (PLBundler and a reformulation of RTScheduler) were added without incident.

In the third stage we physically partitioned the connector expertise and removed it from the compiler source for automatic selection and inclusion in a generated compiler. This allows us to selectively generate versions of compilers with different configurations of connectors. An alpha version of this third-generation compiler will be publicly available in March 1996. Soon after, we will create new connector experts to further test this separation and to diversify our selection of connectors. This will help to verify that connector expertise can be completely localized and created separately from the compiler.

In an upcoming fourth stage, when the expertise for a connector is well understood, we anticipate developing an abstract notation for specifying the expertise from which we will be able to automatically generate it. We also expect analysis of the third prototype to provide the information necessary for adding support for connectors with composite implementations and user-defined connectors in future generations of the compiler.

3.3. What an Expert Knows

In order to generate the intermediate products required to realize connectors, the UniCon compiler requires a considerable body of information about each connector type. The compiler localizes this information into experts; each connector type has its own expert. The body of information takes many forms, as described in Section 3.2. The categories of knowledge in an expert correspond (roughly) to the compiler phases with an additional category corresponding to the graphical editor support.

**Graphical Editor Expertise**

The graphical editor associates an icon with each connector type. Currently, the expertise for each of these icons is captured as a Scheme routine that calls the appropriate drawing commands.

The graphical editor expertise for each connector type also consists of a standard template to assist the designer in creating new connectors of the type.

**Syntax and Semantics Expertise**

Connector experts contain support for syntax checking not performed by the language parser. Some syntax checks are common across all connectors; others are specific to the connector type. The information needed for the common checks is well-understood and the same for each connector, so these checks are implemented as pre-defined functions that expect connector-specific information in tables. Checks supported in this way include:

- Is the given connector type one of the known connector types?
- Is the given role type one of the known role types?
- Is the given property one of the known properties?

Connector-specific syntax checks are implemented as C source code fragments that are collected in a common function.

- Is the value of the given connector property syntactically correct?
- Is the value of the given role property syntactically correct?

Like syntax checks, some semantic checks are common to all connectors and others are connector-specific. The common semantic checks include:

- Is a role of the given type allowed within a connector of the given type?
- Is the given connector property allowed within a connector of the given type?
- Is the given role property allowed within a role of the given type?
- Is the given property a duplicate within the same list?

---

2The graphical editor is implemented using STk, a Scheme interpreter that comes bundled with Ousterhout’s popular Tk toolkit (http://kaolin.unice.fr/html/STk.html).
• Is the number of players connected to a role consistent with the values of its MinConns and MaxConns properties?

Like syntax checks, some of the information for performing common semantic checks is well-understood, and these checks are relegated to table look-ups in predefined functions. Connector-specific semantic checks examine details about connectors of the specified type, and determine the legality of values of connector properties. These are implemented as C source code fragments in common functions. Connector-specific semantic checks for RTScheduler connectors are described in Section 4.

Syntax Tree Building Expertise

During the syntax tree building phase, the UniCon compiler populates attributed syntax tree nodes with attributes synthesized from the parsed input. Each connector expert requires its own set of attributes to be synthesized. This expertise takes the form of C source code fragments added to the function that synthesizes the required attributes for connectors.

Automatic Connection Expertise

Connecting components with many players can be quite tedious, especially when the correspondence is obvious (e.g., RoutineCall and RoutineDef players with the same names and signatures). To simplify these cases, some experts support automatic connection of unconnected players. The knowledge of how to perform automatic connections is a connector-specific part of an expert and is embodied in the form of C source code fragments.

Analysis Expertise

Some parts of a design lend themselves to certain types of analyses. For example, the execution time, priority, and period properties of a set of schedulable processes can be analyzed using rate monotonic analysis (RMA) to see if they will all meet their deadlines [KRPOH93]. The knowledge of how to perform applicable analyses is contained in the connector experts. The expertise includes which data to analyze, how to format it, how to invoke the analysis tools (if any), and how to obtain, formulate, and deliver the results (and possibly incorporate them into the current UniCon compile session). This knowledge is implemented as C source code fragments added to the analysis phase of the compiler. The RTScheduler connector’s analysis expertise is described in Section 4.

Build Expertise

Each expert contains the knowledge of how to build systems that use connectors of its type. This expertise has three parts: how to build the intermediate products necessary to realize the connectors in the target system (e.g., how to create the pipeline initializer program), the knowledge contained in the products (e.g., how to create and initialize a pipeline in the Unix environment), and how to incorporate the intermediate products into the final system (e.g., how to compile and link “glue-code” for a remote procedure call into a given binary executable). The build expertise comprises the bulk of the expert; it is specific to each connector type and is implemented as C source code. As an illustration, Section 4 elaborates the build expertise for the RTScheduler connector.

3.4. Incorporating Connector Expertise in the UniCon Processor

Figure 3 depicts the architecture of the UniCon compilation system. Architectural descriptions are stored in a common form. They can be edited (interchangeably) by any of a variety of editors, currently in batch style by a simple ASCII editor and interactively via a graphical box-and-line drawing editor. Syntactic and semantic checks are performed by an analyzer. When a system construction is requested, the parsed and analyzed UniCon is converted by a builder to a construction script that is executed by Odin. This step may also involve the creation of new code or components to support connector “glue”. Since UniCon’s role ends with the configuration instructions, any initialization or reconfiguration directives must be incorporated in the system that Odin builds.

Connector expertise is used in all stages of the UniCon compiler. For example, connector-specific icons are used in the graphical editor. Code fragments that perform syntax and semantic checks are used by the analyzer to check for correctness of UniCon definitions. The analyzer also uses connector-specific code fragments to perform automatic connections of unconnected components and analyses (e.g., rate monotonic analysis), where applicable. The builder contains expertise to correctly generate any “glue” to complete a connection and the construction specification that will
eventually be used to realize the connection in an executable version of a system.

The expertise for each connector must be created manually. However, once the expertise for a particular connector is created, it is inserted into the UniCon compiler automatically via a compiler generator. The compiler generator builds the UniCon compiler from a specification of connectors, so it is possible to construct a version with any combination of the available built-in connectors that the UniCon language supports.

In the future, we hope to be able to codify the connector expertise such that semi-formal specification of new connectors would be possible, making the expertise automatically generatable as well.

4. Example: The RTScheduler Expert

To illustrate the requirements for connector-specific expertise, we discuss the RTScheduler connector in more detail. This example illustrates the syntax and semantic checking expertise of the connector expert, the UniCon capability of performing an analysis on the elements of a connection (in this case, a RMA), and the generation of code necessary to initialize the real-time scheduler in the operating system at run-time prior to system invocation. It is interesting to note that the entire size of the RTScheduler connector expert is roughly 110K bytes of information, contained in 26 separate files.

Graphical Editor Expertise

The icon for the RTScheduler connector is a pocket watch as depicted in Figure 1.

Syntax and Semantic Checking Expertise

The RTScheduler expert performs the following connector-specific syntax checks:

- The Algorithm and Processor properties must have string values.
- A Trace property must have as its value a semicolon-separated list of at least two qualified identifiers of the form a.b.c.

The RTScheduler expert also performs the following connector-specific semantic checks:

- An Algorithm property must have a value of “rate monotonic,” “time sharing,” “round robin fixed priority,” “fifo fixed priority,” “deadline monotonic,” or “earliest deadline first,” ignoring case.
- In the value of a Trace property, the first item must name a TriggerDef from a periodic schedulable process, and each subsequent item must name a SegmentDef. Each of these items must be (a) in a component that has already been instantiated and (b) connected to the Load role of the RTScheduler connector in which the Trace property appears.
- Each RTScheduler connector must be targeted for a different processor.
- The RTScheduler connector’s Processor property should match the Processor property of each process scheduled by the connector. (Warning only.)
- Each instantiated schedulable process must be connected to exactly one RTScheduler connector.

Syntax Tree Building Expertise

For the RTScheduler connector, the syntax tree building expertise consists of populating the internal syntax tree nodes associated with the players in the Trace property with pointers to the player definitions to which they refer.

Automatic Connection Expertise

No automatic connection of unconnected RTLoad players to RTScheduler connectors is supported.

Analysis Expertise

In the system analysis phase, if the compiler encounters an RTScheduler connector that has an associated “rate
monotonic” scheduling algorithm, it performs a RMA of
the associated processes. This currently consists of
generating a file containing the period, priority, and
execution time information for each schedulable
process, as well as the event trace information
describing the route that remote procedure calls take
through the processes. This file is transmitted to a RMA
tool that analyzes the schedulability of the set of
processes and returns a simple result indicating whether
or not each process will meet its deadline.

Build Expertise

The RTScheduler expert contains three types of build
knowledge. The first is how to initialize a system of
schedulable processes in the Real-Time Mach
environment. Real-time scheduling connectors are
realized by the interaction of schedulable processes
competing for a processor resource. This interaction is
governed by the period and priority properties of each
process and the scheduling algorithm of the processor.
Initialization of a system of schedulable processes
requires first setting the scheduling algorithm of the
processor in the kernel, then starting the processes in the
right order. Each schedulable process must perform its
own initialization step before performing its main
function. During this step, the process registers its period
and priority information with the kernel. As each process
is initialized, the kernel schedules it to run according to
the set of periods and priorities of all initialized
processes.

The second type of knowledge is how to generate the
products that perform system initialization. For each
RTScheduler connector, UniCon extracts the algorithm
information from the attributed syntax tree created
during parsing and generates a C source code module to
initialize the scheduler in the kernel at runtime. UniCon
also produces a Unix shell script for each connector that
invokes the scheduler initialization program and starts
the schedulable processes. For each schedulable process
in an RTScheduler connector, UniCon extracts the
period and priority information from the attributed syntax
tree and creates a process initialization C source code
fragment that registers the period and priority of the
process with the scheduler in the kernel at runtime.

The third type of knowledge is how to transform the C
source code module that performs scheduler
initialization into a program executable, and how to
incorporate the C source code fragment that performs
process initialization into the schedulable process
executable. This knowledge is realized as Odin
construction instructions that are executed to build the
final system.

5. Related Work

Notations for describing the configuration of software
systems has a long history. In 1975, DeRemer and Kron
[DK76] created a notation for describing the structure of
module-based programs, called a module
interconnection language (MIL). 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
[MMS79] and Ada [CE78], and have provided a base
from which to support software construction [Th76],
version control [Co79], system families [Ti79], and
dynamic configuration [MKS89]. Enough examples are
available to develop models of the design space [Pe87,
PN86].

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 lan-
guage, modules can be dynamically instantiated and
bound at runtime [MDK93]. Polygen [CP91] augments a
module interconnection language with an inference
type 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 [Pu90].
This kind of system requires expanding the notion of a
MIL to include specifics about a module's imple-
mentation, such as its programming language, its hard-
ware/operating system platform, and the communication media needed to access it. These configuration notations have recently matured enough to describe both statically and dynamically structured distributed systems [MDK94]. More recently, languages such as UniCon for describing system architectures have started to emerge [Gar95].

6. Conclusion
Our objective is to support the abstractions actually used by software designers to describe the architectures of their software systems. These abstractions are at a considerably higher level than the code. The connectors, or abstractions for interaction mechanisms, present particular problems, as they are often encoded implicitly and diffusely. As an initial step we are developing UniCon, an architecture description language that provides a single set of abstractions and notations to support a wide variety of the component interaction mechanisms commonly provided by languages and operating systems. We are especially concerned with supporting these mechanisms in a uniform way and with minimizing the amount and variety of mechanism-specific information a developer needs to understand.

The implementation task for connectors is harder than conventional compilation, because the connectors are realized not by discrete units of code, but by a variety of actions at construction and initialization time as well as code that is mingled with code devised for other purposes. We have shown how a conventional compiler organization can be extended to address this need. In addition, we have organized the connector-specific expertise to simplify extendibility.

Our strategy of progressive codification has allowed us to gain experience incrementally. Initial explorations with a few connectors helped us understand how to handle the diffuse representation and deal with aspects of connectors that have no concrete realization until execution time. This provided enough experience to begin identifying the kinds of information that must be included in a connector “expert”; this in turn guided reorganization of the compiler in preparation for fully isolating the expertise and enabling easy addition of new connectors.

In the near future we expect to extend the set of supported connectors, provide for configuring versions of UniCon that support selected sets of connectors, and move toward the ability to define new connectors within UniCon.

Acknowledgments
Our collaborators at Carnegie Mellon, especially David Garlan, have provided critical and stimulating feedback. The RTScheduler connector arose from a collaboration with Ragunathan Rajkumar from Carnegie Mellon’s Software Engineering Institute.

This research was supported by the Carnegie Mellon University School of Computer Science, 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. 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 any of the sponsors.

References


