Abstractions and Implementations for Architectural Connections

Mary Shaw[1], Robert DeLine, Gregory Zelesnik

Computer Science Department
Carnegie Mellon University
Pittsburgh PA 15213

November 29, 1995

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]For further information: Electronic mail contact address: mary.shaw@cs.cmu.edu; Web page for current status: http://www.cs.cmu.edu/~Vit/

1. *Introduction*

Software developers frequently describe their designs as sets 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, 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 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 the parts of the implementation that represent particular components, because components are usually manifest in the source code. However, identifying the part of the code that corresponds to a connection is much more difficult, as this code is often diffuse, implicit, or mingled with code with different overt functions. In addition, the code may not capture at all the designer's abstraction, 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 effect of implicit connector information is that it hides the intended abstractions about relations between components. A labeled line between boxes in an architectural diagram represents an abstraction about the system, such as a protocol or a shared representation. 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 heterogeneous and incompatible packaging, and serious integration problems may result. 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 and describes the connectors currently supported in UniCon. Section 3 describes the requirements that first-class connectors impose on the UniCon compiler and the range of implementation strategies that result. Section 4 summarizes our experience.

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, called *players*, through which the component can interact with the outside world. Each player is typed and may list properties that further specify the player (such as the PortBinding and Signature attributes above). 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, called *roles*, at which the connector can mediate the interaction among components. Each role is typed and optionally lists attributes 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 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 (e.g., parts and scripts) 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.

A software system can be represented interchangeably as graphics or text in UniCon. Figure 1 shows a real-time system involving a client and server that communicate via RPC as well as competing for real-time response from the processor. Further properties are provided as details associated with each component and connector; these are inspected and edited dynamically. In the textual form, as shown in Figure 2, a flatter information representation exposes more at the top level.

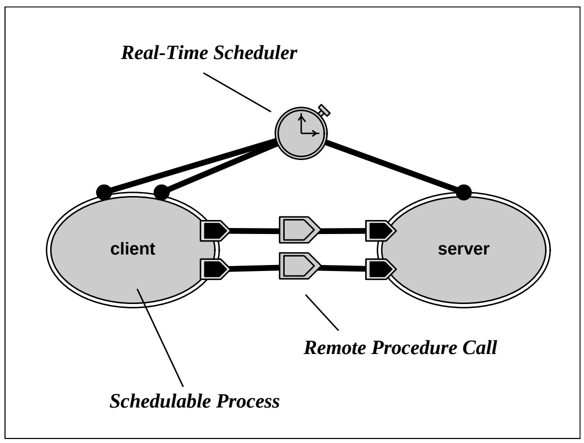


Figure 1. Real-time client-server system with two schedulable tasks sharing a computing resource. The tasks also interact via remote procedure call.

In the example of Figure 1, the two processes interact both through remote procedure calls and by competing for real-time response. The RPC interactions are mediated by the RTM-remote-proc-call connector; the real-time interaction is mediated by the RTM-realtime-sched 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. Figure 2 shows a representative sample.

Component type SchedProcess provides an abstraction for processes that must meet real-time deadlines and must be scheduled accordingly by the real-time operating system. These processes may be periodic or aperiodic. Real-time applications use components of type SchedProcess 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

```
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)
         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
         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
```

```
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 ("new_type *"; "void")
         end timeget
      player timeshow is RPCCall
         SIGNATURE ("void"; "void")
         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
```

Figure 2: Textual form for two components and a connector of example of Figure 1.

from among six scheduling algorithms. If the algorithm rate_monotonic is selected, UniCon invokes an external 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.

2.2. Connector Abstractions Supported by UniCon

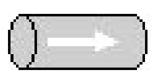
UniCon's types are the most important carriers of abstractions. A connector's type, for example, indicates which roles must be satisfied for the connector to operate properly, together with the types of players that are eligible to play the roles (and which component types may define those players). 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.

Several broad classes of connectors are currently supported, 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.

For concreteness, the remainder of this section describes UniCon's current connectors. For each, we give informal descriptions of the intuition, the properties of the connector, and the roles associated with the connector.

Pipe Connector

Informal Description: The Unix abstraction for pipe, i.e. a bounded queue of bytes that are produced at a source and consumed at a sink. Also supports interactions between pipes and files, choosing the correct Unix implementation.

Icon: pipe section 

Properties: *PipeType*, the kind of Unix pipe. Possible values Named, Unnamed

Roles: *Source*

Description: the source end of the pipe
Accepts player types: *StreamOut* of component Filter; *ReadNext* of component SeqFile
Properties: *MinConns*, minimum number of connections. Integer values, default 1
MaxConns, maximum number of connections. Integer values, default 1

Sink

Description: the sink end of the pipe
Accepts player types: *StreamIn* of component Filter; *WriteNext* of component SeqFile
Properties: *MinConns*, *MaxConns*, as for Source

ProcedureCall Connector

Informal Description: The architectural abstraction corresponding to the procedure call of standard programming languages. Requires signatures (eventually pre/post conditions) in the RoutineDef and RoutineCall players to match; if they don't, requests remediation. Supports renaming.

Icon: blunt arrowhead

Roles: *Definer*

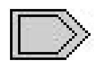
Description: role played by the procedure definition
Accepts player types: *RoutineDef* of component Computation or Module
Properties: *MinConns*, minimum number of definitions allowed. Integer, must be 1
MaxConns, maximum number of definitions allowed. Integer, must be 1

Caller

Description: the role played by the procedure call
Accepts player types: *RoutineCall* of component Computation or Module
Properties: *MinConns*, minimum number of callers allowed. Integer, default 1
MaxConns, maximum number of callers allowed. Integer, default many

RemoteProcCall Connector

Informal Description: The abstraction for the remote procedure call facility supplied by the operating system. Requires signatures and eventually pre/post conditions in the RPCDef and RPCCall players to match. RemoteProcCall connectors require much more UniCon support than ProcedureCall connectors, as they must establish communication paths between processes.



Icon: bordered blunt arrowhead

Roles: *Definer*

Description: role played by the procedure definition
Accepts player types: *RPCDef* of component Process or SchedProcess
Properties: *MinConns, MaxConns*, as for ProcedureCall

Caller

Description: the role played by the procedure call
Accepts player types: *RPCCall* of component Process or SchedProcess
Properties: *MinConns, MaxConns*, as for ProcedureCall

DataAccess Connector

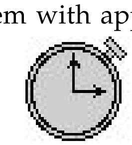
Informal Description: The architectural abstraction corresponding to imported and exported data of conventional programming languages.

Icon: triangle

Roles: *Definer*, essentially similar to *Definer* of *ProcedureCall*
User, essentially similar to *Caller* of *ProcedureCall*

RTScheduler Connector

Informal Description: Mediates competition for processor resources among a set of real-time processes (requires an operating system with appropriate real-time capabilities).



Icon: stopwatch

Properties: *Algorithm*, the scheduling discipline. Possible values: RateMonotonic, TimeSharing, EarliestDeadline, DeadlineMonotonic, RoundRobinFixPriority, FIFOFixPriority
Processor, the name of the processor on which this set of processes will run
Trace, a path through the real-time code and the trigger that invokes it

Roles: *Load*

Description: the role played by a real-time load on a processor
Accepts player types: *RTLoad* of component SchedProcess
Properties: *MinConns*, minimum number of competing processes. Integer, default 2
MaxConns, maximum number of competing processes. Integer, default many

PLBundler Connector

Informal Description: A composite abstraction for matching definitions and uses of a collection of procedures and data. It allows multiple procedure and data definitions and uses to be matched with a single abstraction. Supports renaming.

Icon: chain links 

Properties: *Match*, the correspondences between individual definitions in the bundles. Values are sets of pairs of names.

Roles: *Participant*

Description: a set of definitions and uses to take part in the linkage
Accepts player types: *PLBundle* of component Computation, Module, or SharedData
Properties: *MinConns*, minimum number of bundles to match. Integer, default 2
MaxConns, maximum number of bundles to match. Integer, default many

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 blocks of object code. Connectors are rather translated into a variety of different intermediate and final products that serve to realize the target system during system construction.

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 program executables 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 continuing 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 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 actual code, analysis, configuration of components, and directives to system services. To this end, UniCon produces intermediate products that are used during system construction and initialization as well as final products that persist into execution. These products fall into four major categories:

- 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[2] 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)

Each connector requires a different collection of these intermediate products.

The only products that persist into execution are essentially identical to things now produced by hand. As a result, UniCon imposes no performance cost after runtime initialization.

Pipe connectors are created at initialization time as Unix unnamed or named pipes (Unix fifo files). To create a complex configuration of filters connected by pipes, UniCon generates an initialization routine that creates all of the pipes, then performs the correct collections of forks, port manipulations, and execs in the correct order to establish the UniCon-defined topology in the initialized system. UniCon generates Odin instructions to turn this routine into a program executable.

ProcedureCall, DataAccess, and PLBundler connectors are established at link time. The UniCon compiler generates Odin instructions that invoke the linker with directives that resolve all procedure calls and global data accesses between components. UniCon provides for renaming when procedures and data accesses are connected. When this capability is used, UniCon transforms both names involved in a connection (e.g., the name used by a procedure call in one component and the name used by a procedure definition in another component) to a third, internally generated, unique identifier; these renamings are recorded as a set of C macro definitions used by Odin during the compilation step of a component.

RemoteProcCall connectors require extensive library support that is complicated and tedious to invoke. UniCon generates both the glue code[3] and the process-initialization code to create remote procedure call (RPC) connectors between processes. For the glue code, UniCon generates a specification of the procedure call interface for each connector in an intermediate language. UniCon then generates Odin instructions that (a) invoke a glue-code generator to produce C source code from the specification of the procedure call interface, (b) compile the source code, and (c) link it with the rest of the source modules. In addition to the glue code, UniCon generates special initialization source code for each process making RPCs. This initialization

---

[2]Odin is a system construction utility similar to the "make" utility in Unix [Cl95]. 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@bellcore.com.

[3]Remote procedure calls are implemented by message passing in the target environment. Glue code is necessary for marshaling the arguments in a remote procedure call into a message, passing the message between processes, unmarshaling the arguments in the destination process, and calling the specified procedure (similar actions are required for passing the return value back to the calling procedure).

code registers a process' services and obtains information on the processes that must satisfy its service requests. UniCon also supports renaming for remote procedure calls just as for procedure calls and global data accesses.

RTScheduler connectors are realized by the interaction of processes competing for the processor's computing resource via some scheduling algorithm implemented by a real-time operating system. Real-time scheduling requires each schedulable process to be initialized with certain properties, such as its period and priority. UniCon generates C initialization code for each schedulable process involved in an RTScheduler connector. At runtime, this code creates and initializes the process in the real-time environment with the specified period and priority. In addition, UniCon generates a program executable that initializes the real-time scheduler at runtime. Finally, UniCon generates a Unix shell script that invokes the scheduler-initialization program and starts the schedulable processes at runtime.

UniCon also supports schedulability analysis for real-time systems. When the RTScheduler is invoked, UniCon extracts information from the property lists and creates an input file for a rate monotonic analysis (RMA) tool [SDKRYZ95]. The RMA tool analyzes the data and returns a result indicating whether or not the schedule is achievable.

3.2. Development Strategy

The UniCon implementation is evolving in three stages. In an initial stage, we used ad hoc techniques to support a diverse set of connectors (Pipe, ProcedureCall, DataAccess, RemoteProcCall, 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 UniCon must produce for each type of connector. The first prototype achieved these goals and was frozen in May, 1994.

In the second stage, we are organizing and consolidating our understanding of the knowledge required to handle connector abstractions. Our objective is to identify types of knowledge, tasks, and intermediate products required to implement *every* connector type. We recognize *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.

Analysis of the first prototype revealed commonalities in this knowledge. In the second stage, we are reorganizing the implementation to localize these commonalities. This will make it easier to add new connectors and subsequently move on to the third stage. The second prototype has been stable since October, 1994. It localizes connector-specific expertise. Two new connectors (PLBundler and a complete reformulation of RTScheduler) have been added without incident. The 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). The 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.

We expect analysis of the second prototype to provide the information necessary for adding support for connectors with composite implementations and user-defined connectors in the third-generation 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 UniCon compiler localizes this information into experts; each connector type has its own expert. The body of information takes many forms, as enumerated in Section 3.2.

The categories of knowledge in an expert correspond well to the compiler phases:

- syntax and semantics expertise
- syntax tree building expertise
- automatic connection expertise
- analysis expertise
- build expertise

We now describe an expert's knowledge in terms of these categories.

3.3.1. Syntax and Semantics Expertise

Connector experts contain support for syntax checking not performed by the language parser. Some syntactic 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 known connector types?
- Is the given role type one of the known role types?
- Is the given attribute one of the known attributes?

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

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

Like syntactic 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 attribute allowed within a connector of the given type?

- Is the given role attribute allowed within a role of the given type?
- Is the given attribute-value pair specification a duplicate within the same list?
- Is the number of players connected to a role consistent with its MinConns and MaxConns attribute values?

Like syntactic 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 attributes. These are implemented as C source code fragments in common functions. Connector-specific semantic checks for Pipe and RTScheduler connectors are described in Section 3.4.

The graphical user interface tags the line for each connector with an icon determined by the connector's type. Currently, the expertise for each of these icons is captured as a Scheme[4] routine that calls the appropriate drawing commands.

3.3.2. 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 protocol attributes.

3.3.3. 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. A specific example is provided in Section 3.4.1 for the Pipe connector.

3.3.4. Analysis Expertise

Some parts of a design lend themselves to certain types of analyses. For example, the execution time, priority, and period attributes 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. An example of analysis expertise is discussed in Section 3.4.2 for the RTScheduler connector.

---

[4] The graphical user interface is implemented using STk, a Scheme interpreter that bundles Ousterhout's Tk toolkit (http://kaolin.unice.fr/html/STk.html).

3.3.5. 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), 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 process). The build expertise comprises the bulk of the expert; it is specific to each connector type and is implemented as C source code. Section 3.4 elaborates the build expertise for Pipe and RTScheduler connectors.

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.

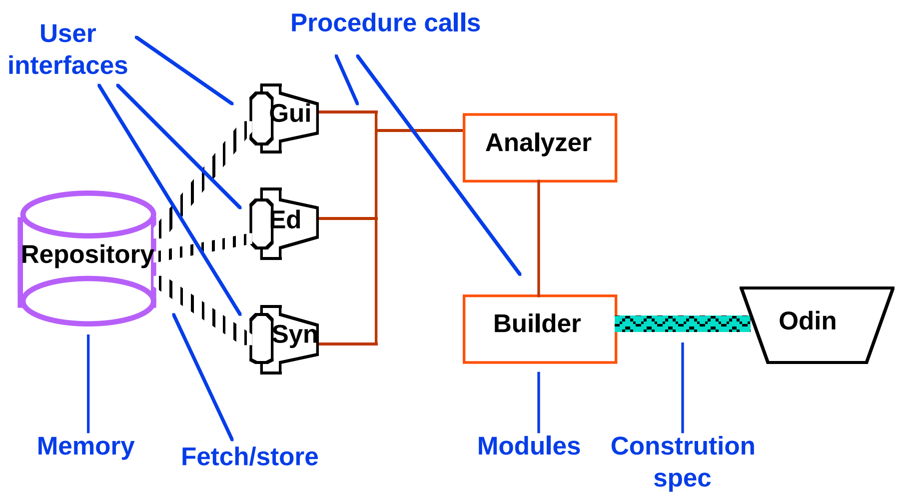


Figure 3: Architecture of UniCon compiler

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 semantics 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 builtin 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.

3.5. Examples

To illustrate the requirements for connector-specific expertise, we discuss two connectors in more detail, Unix pipes and real-time schedulers. The Pipe example illustrates how UniCon produces run-time system initialization code for a pipe-and-filter application as part of the system construction process. The RTScheduler example illustrates the production of "glue" code necessary to realize a remote procedure call at runtime; it also illustrates the creation of code to initialize the real-time scheduler in the operating system at run-time prior to system invocation. Lastly, the RTScheduler example illustrates the UniCon capability of performing an analysis on the elements of a connection (in this case, a RMA).

3.5.1 The Pipe Expert

Syntax and Semantic Checking Expertise

In addition to the common syntactic checks discussed in Section 3.3, the Pipe expert checks that the value of the PipeType attribute is a string value. The Pipe expert also checks that the value of the PipeType attribute is either "named" or "unnamed". The Pipe expert also enforces a semantic rule: a Pipe connector may connect exactly two filters or a filter and a file, but not two files.

Automatic Connection Expertise

If a filter in a component implementation is found to have an unconnected StreamIn or StreamOut player, the Pipe expert uses the following strategy to try to make an automatic connection:

- The compiler searches for a Pipe connector whose appropriate role (the Sink role for an unconnected StreamIn player, the Source role for a StreamOut player) can still accept a connection, and whose opposite role is already connected.
- If the first step fails to yield a connection and the unconnected player is of type StreamOut with a port-binding of 2 (i.e., Unix standard error), the compiler tries to bind the player to a StreamOut player with port-binding 2 in the component's interface.
- If neither step above applies or yields a binding, then the compiler connects the player to the Unix file "/dev/null."

Analysis Expertise

UniCon does not currently support any analyses related to pipes. If we had a tool to analyze throughput in a pipeline, we make it available to the Pipe expert. This might involve adding an attribute for specifying throughput information for a given pipe, collecting the information for each pipe, invoking the tool, and incorporating the results into the compilation.

Build Expertise

The Pipe expert contains two types of build knowledge. The first is how to create a pipe--filter system in the Unix environment. This knowledge is incorporated in the pipeline initialization code generated by the Pipe expert. During runtime initialization, the initializer creates all of the pipes, then "forks" itself once for each filter in the system. Each child process, using a unique index, performs look-ups in topology tables (described below) to connect itself to the correct pipes on the correct Unix file descriptors in the right order[5]. When each child has hooked itself up properly, it "execs" the correct executable program.

The second type of knowledge consists of how to construct the pipeline initializer program. UniCon extracts the pipe-filter topology from the attributed syntax tree created during parsing, then maps the information to a set of tables used as input to the pipeline initializer. UniCon then generates the C source code containing the initialization code and the topology tables and transforms it into an executable program using Odin.

3.5.2 The RTScheduler Expert

Syntax and Semantic Checking Expertise

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

- An Algorithm attribute must have a string value.
- A Processor attribute must have a string value.
- A Trace attribute must have as its value a comma-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 attribute 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 attribute, the first item must name a Trigger from a periodic schedulable process, and each subsequent item must name a Segment. 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 attribute appears.
- Each RTScheduler connector must be targeted for a distinct processor.

---

[5]Order matters when named pipes are used. A process suspends after opening one half of a named pipe until the other half is opened by another process. This opens the possibility of deadlock. The pipeline initializer gurantees that child processes open named pipes in an order that prevents deadlock.

- The RTScheduler connector's Processor attribute should match the Processor attribute of each processes scheduled by the connector. (Warning only.)
- Each instantiated schedulable process must be connected to exactly one RTScheduler connector.

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 an 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 attributes 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 of 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 instructions that are executed to build the final system.

4 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 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 language, modules can be dynamically instantiated and bound at runtime [MDK93]. Polygen [CP91] 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 [Pu90]. 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. 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].

5 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

[AG94] Robert Allen and David Garlan. Formalizing Architectural Connection. In *Proc Sixteenth International Conference on Software Engineering*, 1993.

[Co79] L. W. Cooprider. *The Representation of Families of Software Systems*. PhD Thesis, Carnegie Mellon University. April 1979.

[CE78] SARA Aided Design of Software for Concurrent Systems. *Proc. National Computer Conference*, 1978.

[Cl95] Goeffrey H. Clemm. The Odin System. *Proc. 5th Software Configuration Management Workshop* (SCM5), April 1995.

[CP91] 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.

[DK76] 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.

[Gar95] David Garlan (ed) "First International Workshop on Architectures for Software Systems, Workshop Summary". *ACM Software Engineering Notes*, vol 20, no 3,July 1995, pp. 84-89.

[GS93] 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. 1, World Scientific Publishing Company, 1993, pp.1-39.

[KRPOH93] 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.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73-82, March 1993.

[MDK94] J. Magee, N. Dulay, and J. Kramer. A constructive development evironment for parallel and distributed programs. *Proc. Second International Workshop on Configurable Distributed Systems*, March 1994.

[MKS89] J. Magee, J. Kramer, and M. Sloman. Constructing distributed systems in CONIC. *IEEE Tr. on Software Engineering*, SE-15(6):663-675, 1989.

[MMS79] J. G. Mitchell, W. Maybury, and R. E. Sweet, *Mesa Language Manual*. Tech. Report CSL-79-3, Xerox Corporation, Palo Alto Research Center, April 1979.

[Pe87] Dewayne E. Perry. Software Interconnection Models. In *Proc. Ninth International Conference on Software Engineering,* IEEE Computer Society Press, March 1987.

[PN86] R. Prieto-Diaz and J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4), November 1986, pp. 307-333.

[Pu90] James Purtilo. "The Polylith Software Bus." Dept. of Computer Science, Univ. Maryland, Tech. Rep. 2469, 1990.

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

[SDKRYZ95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, Gregory Zelesnik. Abstractions for Software Architectures and Tools to Support Them. *IEEE Tr. on Software Engineering*, 21(4): 314-335, April 1995.

[Sh93] Mary Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. *Proc Workshop on Studies of Software Design 1993*, to be published by Springer-Verlag 1995.

[Sh95] Mary Shaw. Architectural Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging. *Proc SSR'95: Symposium on Software Reuse*, April 1995.

[Th76] J. W. Thomas. *Module Interconnection in Programming Systems Supporting Abstraction*. PhD Thesis, Brown University. June, 1976.

[Ti79] Walter F. Tichy. "Software Development Control Based on Module Interconnection". *Proc. 4th International Conference on Software Engineering*, Munich, 1979, pp. 29-41.