Proceedings of Eighth International Conference on Software Specification and Design (IWSSD-8), March 1996

A Case Study in Architectural Modelling: The AEGIS System*

Robert Allen David Garlan
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Software architecture is receiving increasingly attention as a critical design level for software systems. However, the current practice of architectural description is largely informal and ad hoc, with the consequence that architectural documents serve as a poor communication mechanism, are difficult to analyze, and may have very little relationship to the implemented system. In an attempt to address these problems several researchers have experimented with formalisms for architectural specification and modelling. One such formalism is WRIGHT. *In this paper we show how* WRIGHT *can be used to provide insight into an architectural design by modelling a prototype implementation of part of the AEGIS Weapons System.*

1 Introduction

A critical aspect of any complex software system is its architecture. At an architectural level of design a system is typically described as a composition of high-level, interacting components. Components represent a system's main computational elements and data stores: clients, servers, filters, databases, etc. Interactions between these elements range from the simple and generic (e.g., procedure call, pipes, shared data access) to the complex and domain-specific (e.g., implicit invocation mechanisms, client-server protocols, database protocols).

The description of a system at an architectural level of design is important for several reasons. Perhaps the most significant is that an architectural description makes a complex system intellectually tractable by characterizing it at a high level of abstraction. In particular, the architectural design exposes the top-level design decisions and permits a designer to reason about satisfaction of system requirements in terms of assignment of functionality to design elements.

For example, for a system in which data throughput is a key issue, an appropriate architectural design would allow the software architect to make system-wide estimates based on values of the throughputs for the individual components. Other issues relevant to this level of design include organization of a system as a composition of components; global control structures; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives.

Unfortunately, the current state of the practice is to use informal, diagrammatic notations (such as box-and-line diagrams) and idiomatic characterizations (such as "client-server organization," "layered system," or "blackboard architecture"). The meanings of such diagrams and phrases, if they are given meanings at all, are typically established by informal convention among a small set of developers. (E.g., "in this diagram a line means a pipe.") This relative informality leads to architectural designs that are inherently ambiguous, difficult to analyze, and hard to mechanize.

To address these problems several researchers are investigating the application of formalisms for architectural specification and modelling. The general thrust of these efforts is to investigate the applicability of existing formal models (sometimes with new surface syntax) to the problem of characterizing and reasoning about software architectures.

An important challenge for this research community is to evaluate these languages and to understand their respective strengths and weaknesses. In the past, software specification languages faced similar needs. One mechanism for comparison has been the use of benchmark problems. Examples from the formal specification community include the library problem, the lift problem, the package router, etc.

What are the comparable problems for architecture? One candidate problem is the "AEGIS Weapons System." AEGIS is a large software system used to control ships for the US Navy. In 1994 a part of the system was used by the ARPA Prototech community as a vehicle to demonstrate their prototyping languages. While (architecturally speaking) the constructed system was relatively simple – less than a dozen architectural components – during the course of construction, it raised a surprisingly number of thorny architectural problems for the system implementors and integrators. The challenge for architectural specification languages, then becomes: could these problems have been detected and/or resolved through appropriate descriptive

*The research reported here was sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330; by National Science Foundation Grant CCR-9357792. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Wright Laboratory, the US Department of Defense, the United States Government, the National Science Foundation. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

formalisms?

In this paper we pick up the gauntlet. Specifically, we look at ways in which the WRIGHT architectural specification language can be used to characterize and reason about the AEGIS architecture. We begin with a brief characterization of related work. Next we outline AEGIS and the challenges that it represents. We then provide a naive architectural description of the architecture, and show how architectural formalism helps expose and resolve some of the architectural problems that arose in the course of implementing the system. We finish by providing a revised, more accurate description of the final system.

2 Related work

Our approach to architectural specification is based on the use of CSP to model the behavior of the components and their interactions in an architectural design. A number of other formalisms have been proposed for modelling architectures of software systems. Inverardi and Wolf have used the Chemical Abstract Machine [BB92] as a formal basis for architectural description [IW95]. Architectural elements are represented by "molecules" and architectural interaction by "reactions." Reactions operate as a set of rewrite rules, which determine how the computations proceed over time. We understand that this formalism has been applied to the AEGIS problem.

In their work on architectures for distributed systems, Magee and Kramer have used the π-calculus to model the dynamic aspects of architectures described in the Darwin language [MK95]. Their work can be viewed as a good example of formalization for a particular style (embodied in Darwin) using a semantic model different than the one we use in this paper.

In their investigations of architectural refinement, Moriconi and his colleagues have characterized styles as theories in first order predicate logic and with Lamport's TLA Actions [MQR95]. The latter treatment is consistent with our approach, although it is based on a somewhat different formal model, and is more concerned with understanding the relationship between different architectural styles than with the expressiveness of an architectural formalism for a given system.

Rapide [L$^+$95] is a module description language, whose interface model is based on partially ordered event sets and event patterns. Rapide's main advantage is that it can be executed to produce event traces that can be examined for violation of interaction invariants. This is in contrast to Wright, which provides a stronger basis for static analysis. Rapide has been used to model the AEGIS System, and in fact was one of the original prototyping languages applied to the problem.

In earlier work the authors and other colleagues have used Z to model architectural style [AG92, GN91, AAG93]. While this work demonstrated that many properties of architecture can be handled in a set-theoretic context, we also found that Z was a poorly matched to the problem of capturing the dynamic behavior of architectural systems. It was this insight (among other things) that led us to develop WRIGHT.

Currently, it is difficult to compare these different formalisms, except in a very abstract way. In fact, it is an open and challenging research problem in itself to determine which of the proposed formalisms are most appropriate for architectural specification. However, one of the ways in which this evaluation can take place is through the use of shared case studies. It is the purpose of this paper to contribute to this process: by exhibiting a case study of a non-trivial system, one that others are also investigating, we hope to provide a more concrete basis for comparison and discussion by the architecture community.

The use of model problems for software architecture is not without precedent. Shaw and her colleagues have devised a set of "challenge problems" for architectures, including "compiler", "ATM", "KWIC", "cruise control", "sea buoy", and others [S$^+$94]. This list is an evolving and one and we expect that through efforts like this paper, AEGIS will be added to the list.

3 The AEGIS "Problem"

The problem was first posed by Bob Balzer at an ARPA program meeting in Fall 1994. It was later re-presented as a challenge problem at the 1995 Dagstuhl Workshop on Software Architecture [GPT95].

The AEGIS Weapons System is a large, complex software system that controls many of the defense functions of modern US Navy ships. As described in one DoD report:

> The AEGIS Weapons Systems (AWS) is an extensive array of sensors and weapons designed to defend a battle group against air, surface and subsurface threats. These weapons are controlled through a large number of control consoles, which provide a wide variety of tactical decision aids to the crew. To manage complexity, the crew can preset conditions under which automated or semi-automated responses occur. This capability is generally referred to as doctrine.

The motivation for using AEGIS as a challenge problem arose through a demonstration exercise of the ARPA Prototyping Technology Program in 1993. Engineers on the real AEGIS system provided a design for a part of the system that takes monitored sensor data about moving objects near the ship, and decides what actions to take. To do this the system must resolve the "tracks" of moving objects against its geometrical model of the ship and nearby entities.

An informal description of the proposed architecture of the system is shown in Figure 1. The system consists of seven modules. The Experment Control module provides simulated input from the operator and sensors. Tracking data is sent to the Track Server, which maintains a record of the currently-monitored moving objects (missiles, other planes, submarines, etc.) within its tracking region. The Doctrine Authoring module receives input describing rules of engagement and activation. The GeoServer module takes doctrine information (from the Doctrine Authoring module), and track information (from the Track Server) and based on its own geometric models, determines which tracks intersect which geometric regions. This information (together with track and doctrine information) is fed to the Doctrine Reasoning module, which determines what action should take place. For the purposes of the prototype these actions, as well as other status information is displayed to the user via a Display Server module. The arrows in the figure indicate the direction of information flow.

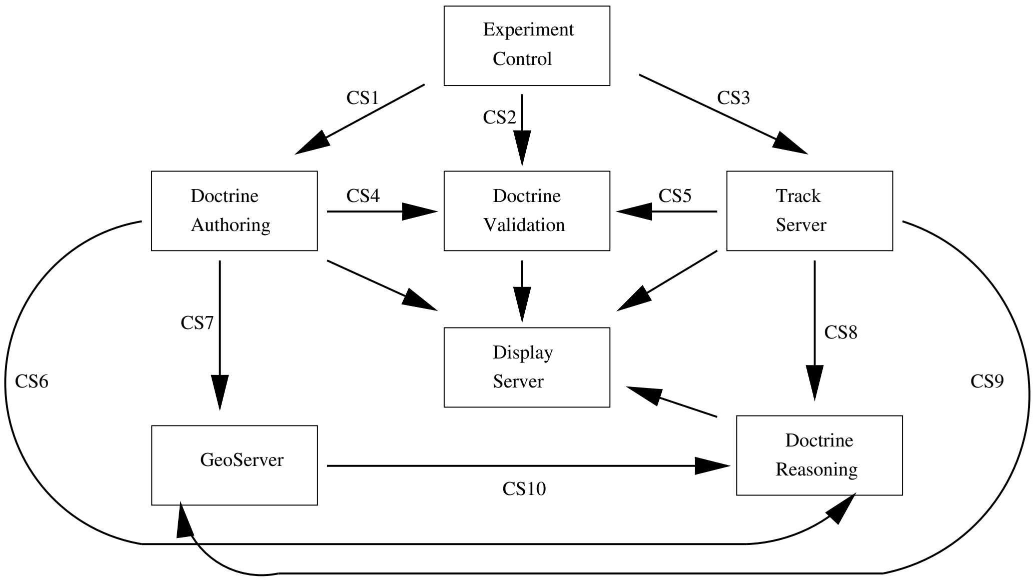

Figure 1: The AEGIS Prototype Architecture

In the Prototech demonstration, each of the research teams in the program was assigned the task of implementing one or more modules of the system. The modules were to be integrated into a running system that could then be demonstrated for the program sponsors. To make this integration possible the teams had to agree on the nature of the architectural connection that they would use. For implementation reasons (they were building on top of Unix with sockets) they initially agreed to use a uniform client-server organization, in which clients requested data from the servers. Thus information would be "pulled" from the top to the bottom of the figure: i.e., clients at the tip of the arrows, and the servers at the tails. Components that have both incoming and outgoing arrows, would act both as a client and a server.

Putting aside internal details of the individual modules, this sounds like a relatively straightforward task. Unfortunately, it turned out to be anything but trivial. First, there were some serious misconceptions about the meaning of client-server interactions. Which party initiated the connection? Was it reestablished after each request? Was the data transferred synchronously? Moreover, there turned out to be restrictions induced by implementation constraints of the modules making it infeasible for certain modules to act both as clients and as servers. Furthermore, the basic design did not account for some advanced monitoring capabilities of the inter-module communication. The net result was that (according to one of the participants) the final integration was something of a nightmare, and the resulting system considerably more complex than had been originally envisioned.

In the remainder of this paper we use the WRIGHT architectural specification language to expose some of these problems. While space does not allow us to treat all of the issues, we will focus on a few key problems – primarily those relating to potential deadlock. We start by characterizing the naive architectural design. Then we show how it must be modified to characterize the "as-built" system.

4 The WRIGHT Notation

Before presenting the AEGIS specification, we provide a brief overview of the WRIGHT notation. (We will assume rudimentary familiarity with CSP [Hoa85].) Details of the semantic model and the supporting toolset can be found elsewhere [AG94b, AG94a].[1]

WRIGHT describes the architecture of a system as a collection of components interacting via instances of connector types. A simple Client-Server system description is shown in Figure 2. This example shows the three elements of a system description: style declaration, instance declarations, and attachments. The instance declarations and attachments together define a system configuration.

An *architectural style* is a family of systems with a common vocabulary and rules for configuration. A simple style definition is illustrated in Figure 3. This style defines the vocabulary for the system example of Figure 2. (Although not illustrated in this paper a style can also define topological constraints on systems that use the style.)

In WRIGHT, the description of a component has two important parts, the *interface* and the *computation*. An interface consists of a number of *ports*. Each port defines the set of possible interactions in which the component may participate.

A connector represents an interaction among a collection of components. For example, a pipe represents a sequential flow of data between two filters. A WRIGHT description of a connector consists of a set of *roles* and the *glue*. Each role defines the behavior of one participant in the interaction. A

[1] The version of WRIGHT used in this paper differs in minor ways from previously published papers. In particular, this version distinguishes input and output events, and introduces a quantification operator and conditional process expression. These differences are elaborated in this section.

System SimpleExample
Style ClientServer
Instances
s: Server
c: Client
cs: C-S-connector
Attachments
s.provide **as** cs.server;
c.request **as** cs.client
end SimpleExample.

Figure 2: A Simple Client-Server System

Style ClientServer
Component Server
Port Provide *[provide protocol]*
Computation *[Server specification]*
Component Client
Port Request *[request protocol]*
Computation *[Client specification]*
Connector C-S-connector
Role Client *[client protocol]*
Role Server *[server protocol]*
Glue *[glue protocol]*
end ClientServer.

Figure 3: A Simple Client-Server Style

pipe has two roles, the source of data and the recipient. The glue defines how the roles will interact with each other.

Each part of a WRIGHT description – port, role, computation, and glue – is defined using a variant of CSP. For example, a simple client role might be defined as:

Role Client = ($\overline{\text{request}}$ →result?x → Client) ⊓ §

A participant in an interaction repeatedly makes a request and receives a result, or chooses to terminate successfully.

As is partially evident from this example, WRIGHT extends CSP in some minor syntactic ways. First, it distinguishes between *initiating* an event and *observing* an event. An event that is initiated by a process is written with an overbar: The specification of the Client's Request port would use the event $\overline{\text{request}}$ to indicate that it initiates a request. The Server's Provide port, on the other hand, waits for some other component to initiate a request (it *observes* the event), so in its specification this event would be written without an overbar: request.

Second, a special event in WRIGHT is $\surd$, which indicates the successful termination of a computation. Because this event is not a communication event, it is not considered either to be initiated or observed. Typically, use of $\surd$ occurs only in the process that halts immediately after indicating termination: §= $\surd$→STOP.

Third, to permit parameterization of connector and component types, WRIGHT uses a quantification operator: $\forall\, x : S\langle op\rangle P(x)$. This operator constructs a new process based on a process expression and the set S, combining its parts by the operator $\langle op\rangle$. For example, $\forall\, i : \{1,2,3\} \,[]\, P_i = P_1 \,[]\, P_2 \,[]\, P_3$. A special case is $\forall\, x : S \,;\, P(x)$, which is some unspecified sequencing of the processes: $\forall\, x : S \,;\, P(x) = \forall\, x : S \sqcap (P(x) \,;\, \forall\, y : S \setminus \{x\} \,;\, P(y))$.

The final extension is the use of "conditional processes," which we will illustrate in the next section.

As discussed in [AG94b], descriptions of connectors can be used to determine whether the glue constrains the roles enough to guarantee critical properties such as local absence of deadlock. These descriptions can also be used to determine whether a configuration is properly constructed, *e.g.,* whether the interfaces of a component are appropriate for use in a particular role. But these issues are beyond the scope of this paper.

The global behavior of a WRIGHT architecture *system instance* is constructed from the processes introduced by the component and connector types in the style definition. For each component instance, the process specifying the component type's **Computation** is relabelled with the component instances name and placed in parallel (CSP operator ||) with the other component instances. The connector instances influence the components' communication pathways by appearing as similarly relabelled **Glue** processes.

In addition to relabelling the glue processes so that different instances of the same connector have distinct event names, the connector instances' events are also renamed so that the attachments represent a communication pathway. For example, the attachment 's.provide **as** cs.server,' shown in figure 2, would mean that each event with the prefix cs.server (for example cs.server.e) would be renamed to have the prefix s.provide (for example s.provide.e). The net effect of this renaming is that a connector instance that has its roles attached to a particular set of ports synchronizes the events of those ports in the global system behavior. In figure 2, this means that the glue of C-S-connector will synchronize the roles s.provide and c.request, just as we would expect from the attachment and instance declarations.

5 The Naive Specification

As noted earlier, the simple model of the AEGIS system uses a client-server model; a client initiates a data request from a server, which fills the requests of each of its clients as they arrive. But this simple, informal, description brushes a lot of important information under the rug, and leaves us without enough details even to begin a more detailed design. The abstraction doesn't resolve issues such as what protocols are used to make the data request and reply, how termination is signalled, and whether servers must handle multiple requests simultaneously. By characterizing this "naive" architectural description in WRIGHT, we will see how these issues come to the fore.

In WRIGHT we begin an architectural description by characterizing the architectural "style" from which the system is developed. We will develop each of the elements of the architecture as a type, either of port or role, component, or connector. Each of the type definitions will provide a building block from which the particular system instance can be developed.

The smallest building block in a system is a protocol fragment, used to describe either the interface of a component or the constraints on a participant in an interaction protocol (connector). These protocol fragment patterns are introduced as **Process** types.

Process ClientPullT = $\overline{\text{open}} \rightarrow$Operate $\sqcap$ §
    where Operate = $\overline{\text{request}} \rightarrow$result?x$\rightarrow$Operate
        $\sqcap$ Close
      Close = $\overline{\text{close}} \rightarrow$§

Process ServerPushT = open $\rightarrow$Operate $\square$ §
    where Operate = request$\rightarrow \overline{\text{result!x}} \rightarrow$Operate
        $\square$ Close
      Close = close$\rightarrow$§

The ClientPullT is the basic type used for the ports of a client component (and for the role in a connector which will be played by such a component). As we will see in the component and connector declarations, the ClientPullT process indicates that a client will begin by establishing the connection with the open event. After opening the connection, an operational phase is begun, in which the client repeatedly chooses to request data. The client expects to receive exactly one result for each request. At any time, the client may choose to close the connection, after which the interaction ceases (as indicated by the § process).

The ServerPushT process is the complement of the ClientPullT. The server expects another party to open the connection (it *observes* this event, as indicated by the absence of an overbar). Then, it will repeatedly provide responses to requests until it recognizes a close event, after which it is free to terminate.

In combination (or even singly), these processes would seem to be adequate to define the client-server interaction; each initiated event in one process corresponds to an observed event in the other. The ClientServer connector specification confirms this relation between events; each line of the **Glue** specification indicates the correspondence between a pair of events – when the client initiates an open, the server will observe an open, and so on. This is a very common case in architectural connectors, and many descriptive notations specialize their connector descriptions to it (*e.g.,* [L$^+$95, YS94]). As we will see later, however, there are other cases where there may be more complex relationships, involving partial visibility of events or run-time mechanisms (that are not part of the abstract computation) that require a more complex **Glue**. WRIGHT requires that even the "trivial" glue be spelled out in full, although it could easily be generated automatically.

Connector ClientServer =
  Role Client = ClientPullT
  Role Server = ServerPushT
  Glue = Client.open $\rightarrow \overline{\text{Server.open}} \rightarrow$**Glue**
    $\square$ Client.close$\rightarrow \overline{\text{Server.close}} \rightarrow$**Glue**
    $\square$ Client.request$\rightarrow \overline{\text{Server.request}} \rightarrow$**Glue**
    $\square$ Server.result?x$\rightarrow \overline{\text{Client.result!x}} \rightarrow$**Glue**
    $\square$ §

In a client-server system such as AEGIS, there are three kinds of components: those that act as clients, those that act as servers, and those that combine the two functions. Components of each of these kinds can have different numbers of interfaces (client or server interfaces), and so we represent them by parameterized types.

Component Client($\texttt{numServers}$: 1..) =
  Port Service$_{1..\texttt{numServers}}$ = ClientPullT
  Computation = ($\forall\, x$:1..$\texttt{numServers}$;
      $\overline{\text{Service}_x\text{.open}}$) ; UseOrExit
  where UseOrExit = UseService $\sqcap$ Exit
    UseService = $\forall\, x$:1..$\texttt{numServers}$
      $\sqcap$ $\overline{\text{Service}_x\text{.request}}$
        $\rightarrow$Service$_x$.result?y
        $\rightarrow$UseOrExit
    Exit = ($\forall\, x$: 1..$\texttt{numServers}$; $\overline{\text{Service}_x\text{.close}}$) ;
      §

The Client component type is straightforward. It has complete control over its actions at any time, as long as it obeys the ClientPullT protocol on each of its ports. For simplicity, we assume that it begins by opening each of its connections, and finishes by closing each of them. During the middle phase, the process UseService selects from among its connections to request a new data item. The choice is entirely up to the client, as indicated by the use of non-deterministic choice.

The picture for a server is considerably more complicated (see figure 4). A Server component provides data services to one or more clients. With each client, the server uses the ServerPushT protocol. At any point in the protocol, each client is in one of three states: "Open" (represented by the set O), "Closed" (represented by the set C), or "not yet Open" (all others).

The **Computation** specification shows many of the difficult issues that arise in specifying this architectural style. What mechanisms are available for the server to locate new connections that should be opened? To receive a client's request? When can a newly closed client connection be recognized, and what action should be taken? As a component *type*, the Server specification provides a generic answer to these questions.

The **Computation** shown in figure 4 describes a server that can handle at most one client request at a time. This is indicated by the non-deterministic (or internal) choice among versions of the process ReadFromClient. The server may also choose to wait for an open request from any of a set of clients. This interaction pattern, of selecting a single client for a request or a set of clients for an open, is characteristic of the Unix socket mechanism, which was selected as an implementation base. We can see the consequences of this choice, while abstracting other implementation details, in its effects on the server component type.

Because of the blocking open and request protocol, as well as the requirement that the server eventually handle all requests, the server must keep track of the statuses of the different clients. Some have not yet opened, and the server can wait for them to open; some have opened but not closed, and server can expect either a request or a close from them; and some have closed, and the server must not expect any further action from them. The open and the closed clients are represented by the state variables O and C respectively (those that have never opened are members of the set $(1..\texttt{numClients}) \setminus (O \cup C)$). Given these different statuses, there are four distinct cases for the server, requiring different choices of action: The least constrained case is when there are both open and unopened clients; in this case, the server is free to make any choice of action. If every client has already opened, the server must not wait

Component Server($\texttt{numClients} : 1..$) =
  Port $\text{Client}_{1..\texttt{numClients}}$ = ServerPushT
  Computation = $\text{WaitForClient}_{\{\},\{\}}$
    where $\text{WaitForClient}_{O,C} = \forall x : ((1..\texttt{numClients}) \setminus (O \cup C)) \; [] \; \text{Client}_x.\text{open} \rightarrow \text{DecideNextAction}_{O \cup \{x\}, C}$
      $\text{DecideNextAction}_{O,C} = \text{WaitForClient}_{O,C} \sqcap \forall x : O \sqcap \text{ReadFromClient}_{x,O,C}$
        $O \neq \{\} \wedge O \cup C \neq (1..\texttt{numClients})$
      $\text{DecideNextAction}_{O,C} = \forall x : O \sqcap \text{ReadFromClient}_{x,O,C}$
        $O \neq \{\} \wedge O \cup C = (1..\texttt{numClients})$
      $\text{DecideNextAction}_{\{\},C} = \text{WaitForClient}_{\{\},C}$
        $C \neq (1..\texttt{numClients})$
      $\text{DecideNextAction}_{\{\},(1..\texttt{NumClients})} = \S$
      $\text{ReadFromClient}_{x,O,C} = \text{Client}_x.\text{request} \rightarrow \overline{\text{Client}_x.\text{result!y}} \rightarrow \text{DecideNextAction}_{O,C}$
        $[] \; \text{Client}_x.\text{close} \rightarrow \text{DecideNextAction}_{O \setminus \{x\}, C \cup \{x\}}$

Figure 4: **Component** Server

for a client to open. If no client is open, then the server does not have the option of waiting for a request. Once every client has closed, the only possible action by the server is to terminate.

These various cases are handled by the use of conditional process definitions:

$\text{P}_V = \text{Q}$
  $p(V)$

defines a process P over variables V only when the boolean expression $p(V)$ is true.

The component type MixedComp (figure 5) combines the properties of a Client and a Server. It must deal with open, close, and request events from its clients, but it also has the option of requesting a service from one of its servers at any time.

Now that we have described the basic vocabulary of the naive AEGIS style, we can show the configuration of the testbed system:[2]

Configuration Testbed
Style Aegis
Instances
  ExperimentControl : Server(3)
  DoctrineAuthoring : MixedComp(1,3)
  DoctrineValidation : Client(3)
  TrackServer : MixedComp(1,3)
  GeoServer : MixedComp(2,1)
  DoctrineReasoning : Client(3)
  $\text{CS}_{1..10}$: ClientServer
Attachments
  ExperimentControl.Client **as** CS_1.Server
  DoctrineAuthoring.Service **as** CS_1.Client
  ExperimentControl.Client **as** CS_2.Server
  DoctrineValidation.Service **as** CS_2.Client
  ExperimentControl.Client **as** CS_3.Server
  TrackServer.Service **as** CS_3.Client
  DoctrineAuthoring.Client **as** CS_4.Server
  DoctrineValidation.Service **as** CS_4.Client
  TrackServer.Client **as** CS_5.Server
  DoctrineValidation.Service **as** CS_5.Client
  DoctrineAuthoring.Client **as** CS_6.Server
  DoctrineReasoning.Service **as** CS_6.Client
  DoctineAuthoring.Client **as** CS_7.Server
  GeoServer.Service **as** CS_7.Client
  TrackServer.Client **as** CS_8.Server
  DoctrineReasoning.Service **as** CS_9.Client
  TrackServer.Client **as** CS_9.Server
  GeoServer.Service **as** CS_9.Client
  GeoServer.Client **as** CS_{10}.Server
  DoctrineReasoning.Service **as** CS_{10}.Client
end Testbed.

[2] For reasons that will become clearer later this initial description excludes the DisplayServer. In the next section we include it in the specification.

6 Analyzing and Changing the Specification

The WRIGHT specification described in the previous section is a reasonable and useful description of the architecture of the AEGIS system as it was initially envisioned. The protocol described in the ClientServer connector and the computation patterns covered by the connectors Client, Server, and MixedComp describe the high level design of the system, exposing the computation model and the requirements on the run-time infrastructure for the proposed system.

The specification also exposes a number of shortcomings of the initial design, shortcomings which led to a major reworking of the system and that seriously complicated the final product. The system as it was eventually constructed bore little resemblance to the simple client-server system described above. In this section, we look at some of the issues that arose in the AEGIS design, show how they are exposed by the preceding WRIGHT description, and further show how the solutions found by the AEGIS team can be expressed in WRIGHT, thus ensuring that the architectural description matches the system as built.

6.1 Issue: Direction of Data Flow

One of the issues that is exposed by the formal description of the AEGIS system is that of dataflow. The AEGIS system contains a server, the Display Server that does not supply data, but instead receives it. The protocols described above for the ClientServer connector, which simply encode the default interpretation of client-server interaction, does

Component MixedComp($\texttt{numServers} : 1..; \texttt{numClients} : 1..$) =
Port $\text{Service}_{1..\texttt{numServers}}$ = ClientPullT
Port $\text{Client}_{1..\texttt{numClients}}$ = ServerPushT
Computation = OpenServices ; $\text{WaitForClient}_{\{\},\{\}}$
where $\text{WaitForClient}_{O,C} = \forall\, x : ((1..\texttt{numClients}) \setminus (O \cup C))\ \Box\ \text{Client}_x\text{.open} \rightarrow \text{DecideNextAction}_{O \cup \{x\}, C}$
$\text{DecideNextAction}_{O,C} = \text{WaitForClient}_{O,C} \sqcap \forall\, x : O \sqcap \text{ReadFromClient}_{x,O,C}$
$\sqcap\ (\text{UseService} ; \text{DecideNextAction}_{O,C})$
$O \neq \{\} \wedge O \cup C \neq (1..\texttt{numClients})$
$\text{DecideNextAction}_{O,C} = \forall\, x : O \sqcap \text{ReadFromClient}_{x,O,C} \sqcap (\text{UseService} ; \text{DecideNextAction}_{O,C})$
$O \neq \{\} \wedge O \cup C = (1..\texttt{numClients})$
$\text{DecideNextAction}_{\{\},C} = \text{WaitForClient}_{\{\},C} \sqcap (\text{UseService} ; \text{DecideNextAction}_{\{\},C})$
$C \neq (1..\texttt{numClients})$
$\text{DecideNextAction}_{\{\},(1..\texttt{NumClients})} = (\text{UseService} ; \text{DecideNextAction}_{\{\},(1..\texttt{numClients})}) \sqcap \text{Exit}$
$\text{ReadFromClient}_{x,O,C} = \text{Client}_x\text{.request} \rightarrow (\text{OptionalUseService} ; \overline{\text{Client}_x\text{.result!y}} \rightarrow \text{DecideNextAction}_{O,C})$
$\Box\ \text{Client}_x\text{.close} \rightarrow \text{DecideNextAction}_{O \setminus \{x\}, C \cup \{x\}}$
$\text{UseService} = \forall\, x : (1..\texttt{numServers}) \sqcap \overline{\text{Service}_x\text{.request}} \rightarrow \text{Service}_x\text{.result?y} \rightarrow \S$
$\text{OptionalUseService} = (\text{UseService} ; \text{OptionalUseService}) \sqcap \S$
$\text{OpenServices} = \forall\, x : (1..\texttt{numServers}) ; \overline{\text{Service}_x\text{.open}} \rightarrow \S$
$\text{Exit} = \forall\, x : (1..\texttt{numServers}) ; \overline{\text{Service}_x\text{.close}} \rightarrow \S$

Figure 5: **Component** MixedComp

not handle this situation. Thus, interactions with the Display Server require a second connector, ClientServerPush, along with corresponding new port/role declarations. (The term Push is used to indicate that the client *pushes* data toward the server, rather than pulling it from the server.)

Process ClientPushT = $\overline{\text{open}} \rightarrow \text{Operate} \sqcap \S$
where Operate = $\overline{\text{request!x}} \rightarrow \text{result} \rightarrow \text{Operate}$
$\sqcap$ Close
Close = $\overline{\text{close}} \rightarrow \S$

Process ServerPullT = Open $\rightarrow$Operate $\Box$ §
where Operate = request?x$\rightarrow\overline{\text{result}}\rightarrow$Operate
$\Box$ Close
Close = close$\rightarrow$§

Connector ClientServerPush =
Role Client = ClientPushT
Role Server = ServerPullT
Glue = Client.open $\rightarrow\overline{\text{Server.open}}\rightarrow$**Glue**
$\Box$ Client.close$\rightarrow\overline{\text{Server.close}}\rightarrow$**Glue**
$\Box$ Client.request?x$\rightarrow\overline{\text{Server.request!x}}\rightarrow$**Glue**
$\Box$ Server.result$\rightarrow\overline{\text{Client.result}}\rightarrow$**Glue**
$\Box$ §

6.2 Issue: Potential for Deadlock in Servers

A more serious issue is the potential for deadlock in a system that uses the Server component as described above. Deadlock can arise because the server must, in effect, *guess* which of the clients will be the next one to make a request. A server does not deadlock on its own: If the clients are able to fulfill their obligation to either request or close, then no problems occur. Deadlock can be a problem, however, when more than one client and server are involved. Consider a simplified system topology with two servers, S1 and S2, and two clients C1 and C2, in which both clients interact with both servers. (One such pattern occurs in the system with components DoctrineAuthoring, TrackServer, GeoServer, and DoctrineReasoning.) What happens if client C1 plans to make a request first to S1 and then S2, while client C2 makes a request first to S2 and then S1? If S1 and S2 both guess wrong about which component will make the first request (*i.e.* S1 guesses C2 and S2 guesses C1), then the system will deadlock. Neither service can proceed before the other, since each is waiting for the other client, which is itself waiting for the other server. The problem of guessing which client will be next is exposed in the specification as a non-deterministic choice over the set of request events in the Server (and MixedComp) specification. This indicates that the servers are free to handle any of their clients, excluding the other clients while doing so.

6.2.1 Using Dynamic Connections

The AEGIS designers took three approaches to solving this problem. The first takes advantage of the fact that it is possible to make a deterministic choice over the set of open events. The protocols are changed so that an open event precedes *every* client request:[3]

Process DynamicClientPullT = $\overline{\text{open}} \rightarrow \overline{\text{request}}$
$\rightarrow$result?x$\rightarrow\overline{\text{close}}$
$\rightarrow$DynamicClientPullT
$\sqcap$ §
Process DynamicServerPushT = open $\rightarrow$request
$\rightarrow\overline{\text{result!x}}\rightarrow$close
$\rightarrow$DynamicServerPushT
$\Box$ §

[3] Similar definitions for DynamicClientPushT and DynamicServerPullT, not shown. Also, connectors DClientServer and DClientServerPush are straightforward but omitted for brevity.

These protocols can be used to make a DynamicServer that waits for open events rather than request events:

Component DynamicServer (`numClients` : 1..) =
  Port Client$_{1..\texttt{numClients}}$ = DServerPushT
  Computation = WaitForClient [] §
  where WaitForClient = $\forall\, i$:1..`numClients`
    [] Client$_i$.open
    $\rightarrow$Client$_i$.request
    $\rightarrow\overline{\text{Client}_i\text{.result!x}}$
    $\rightarrow$Client$_i$.close$\rightarrow$**Computation**

This problem can also arise with the service request portion of a MixedComp, and this is solved by *serverizing* a mixed computation. That is, instead of using a ClientPullT to wait for data, the component uses a ServerPullT port to receive notification when data is available.

Component DynamicServerized (`numServers` : 0..;
  `numClients` : 0..) =
  Port Service$_{1..\texttt{numServers}}$ = DServerPullT
  Port Client$_{1..\texttt{numClients}}$ = DServerPushT
  Computation = WaitForService [] WaitForClient [] §
  where WaitForService =
    $\forall\, i$:1..`numServers`
      [] Service$_i$.open$\rightarrow$Service$_i$.request?x
      $\rightarrow\overline{\text{Service}_i\text{.result}}\rightarrow$Service$_i$.close
      $\rightarrow$**Computation**
    WaitForClient =
    $\forall\, i$: 1..`numClients` [] Client$_i$.open
      $\rightarrow$Client$_i$.request
      $\rightarrow\overline{\text{Client}_i\text{.result!x}}$
      $\rightarrow$Client$_i$.close
      $\rightarrow$**Computation**

6.2.2 Avoiding Unnecessary Synchronizations

A second approach to avoiding a server blocking on a request is to permit the server to provide the data asynchronously, in anticipation of a request. This is achieved through the use of an OpenLoopBuffer connection (figure 6), which guarantees that the source of data will never block waiting for the target to become ready. The buffer stores data until the target requests it, or blocks the target until the source makes new data available. (Such a connection is essentially a pipe.)

Notice how the WRIGHT **Glue** mechanism permits this interaction to be described *without modifying the component interfaces*. This connector can replace a ClientServer connector without modifying the data target, or a ClientServerPush connector without modifying the data source.

6.2.3 Multi-threading Components

The third and final approach to avoiding server deadlocks is perhaps the most flexible: to alter the server's implementation mechanism so that it can handle multiple connections at once. If we use a multi-threaded implementation (easily represented in CSP using the || operator), a single component type can use each of the possible interface protocols (see figure 7).[4]

This solution is used for two of the components, TrackServer and ExperimentControl. The instance declarations for these are as follows:

TrackServer : ThreadedMixedComp (3,0,1,0,1,0,0,0)
ExperimentControl : ThreadedMixedComp (1,0,0,0,0,0,0,2)

TrackServer has three static push server ports, one dynamic push server port, and one static pull client port. ExperimentControl has one static push server port and two dynamic push client ports.

6.3 Issue: Instrumenting Communication

Another issue that is not dealt with in the initial specification is that the AEGIS testbed is an experimental system. As such, there is a requirement that the interactions of the system be monitored. This monitoring must not, of course, alter the components as designed, or the data collected would be invalid. This results in the need for instrumented connectors. WRIGHT can represent these easily by adding a new Listener role to each connector specification, and altering the **Glue** to copy data to the new participant. For example, an instrumented ClientServer connector could be described as follows:

Connector InstrumentedClientServer =
  Role Client = ClientPullT
  Role Server = ServerPushT
  Role Listener = data?x $\rightarrow$Listener [] §
  Glue = Client.open $\rightarrow\overline{\text{Server.open}}\rightarrow$**Glue**
    [] Client.close$\rightarrow\overline{\text{Server.close}}\rightarrow$**Glue**
    [] Client.request$\rightarrow\overline{\text{Server.request}}\rightarrow$**Glue**
    [] Server.result?x$\rightarrow\overline{\text{Client.result!x}}$
    $\rightarrow\overline{\text{Listener.data!x}}\rightarrow$**Glue**
    [] §

The same Listener can also be added to any of the other connectors. For example, InstrumentedDClientServerPush:

Connector InstrumentedDClientServerPush =
  Role Client = DClientPushT
  Role Server = DServerPullT
  Role Listener = data?x$\rightarrow$Listener [] §
  Glue = Client.open $\rightarrow\overline{\text{Server.open}}\rightarrow$**Glue**
    [] Client.request?x $\rightarrow\overline{\text{Server.request!x}}$
    $\rightarrow\overline{\text{Listener.data!x}}\rightarrow$**Glue**
    [] Server.result $\rightarrow\overline{\text{Client.result}}\rightarrow$**Glue**
    [] Client.close $\rightarrow\overline{\text{Server.close}}\rightarrow$**Glue**
    [] §

7 The Updated AEGIS System

We can now give the full, new configuration:

Configuration Testbed2
Style Aegis

[4] While CSP makes this look like the simplest solution of all, depending on the implementation base it may require a complex implementation.

Connector OpenLoopBuffer =
  Role Source = ClientPushT
  Role Target = ClientPullT
  Glue = OpenPhase ; Operate$_{\langle\rangle}$
    where OpenPhase = Source.open $\rightarrow$ $\overline{\text{Target.open}}$ $\rightarrow \S$ [] Target.open $\rightarrow$ Source.open $\rightarrow \S$
      Operate$_{\langle\rangle}$ = Source.request?x $\rightarrow$ $\overline{\text{Source.result}}$ $\rightarrow$ Operate$_{\langle x\rangle}$
        [] Target.request $\rightarrow$ WaitForData
      Operate$_{S+\langle x\rangle}$ = Source.request?y $\rightarrow$ $\overline{\text{Source.result}}$ $\rightarrow$ Operate$_{\langle y\rangle+S+\langle x\rangle}$
        [] Target.request $\rightarrow$ $\overline{\text{Target.result!x}}$ $\rightarrow$ Operate$_S$
      WaitForData = Source.request?x $\rightarrow$ $\overline{\text{Target.result!x}}$ $\rightarrow$ $\overline{\text{Source.result}}$ $\rightarrow$ Operate$_{\langle\rangle}$

Figure 6: An Open Loop Buffer

Component ThreadedMixedComp (`numPushServers` :0..; `numPullServers` : 0.. ;
  `numDPushServers` :0..; `numDPullServers` :0..;
  `numPullClients` : 0..; `numPushClients` : 0..;
  `numDPullClients` : 0..; `numDPushClients` : 0..;
) =
  Port PushServer$_{1..\mathtt{numPushServers}}$ = ServerPushT
  Port PullServer$_{1..\mathtt{numPullServers}}$ = ServerPullT
  Port DPushServer$_{1..\mathtt{numDPushServers}}$ = DServerPushT
  Port DPullServer$_{1..\mathtt{numDPullServers}}$ = DServerPullT
  Port PushClient$_{1..\mathtt{numPushClients}}$ = ClientPushT
  Port PullClient$_{1..\mathtt{numPullClients}}$ = ClientPullT
  Port DPushClient$_{1..\mathtt{numDPushClients}}$ = DClientPushT
  Port DPullClient$_{1..\mathtt{numDPullClients}}$ = DClientPullT
  Computation = $\forall\, i$: 1..`numPushServers` || PushServer$_i$**:**ServerPushT
    || $\forall\, i$: 1..`numPullServers` || PullServer$_i$**:**ServerPullT
    || $\forall\, i$: 1..`numDPushServers` || DPushServer$_i$**:**DServerPushT
    || $\forall\, i$: 1..`numDPullServers` || DPullServer$_i$**:**DServerPullT
    || $\forall\, i$: 1..`numPushClients` || PushClient$_i$**:**ClientPushT
    || $\forall\, i$: 1..`numPullClients` || PullClient$_i$**:**ClientPullT
    || $\forall\, i$: 1..`numDPushClients` || DPushClient$_i$**:**DClientPushT
    || $\forall\, i$: 1..`numDPullClients` || DPullClient$_i$**:**DClientPullT

Figure 7: A Multi-threaded Solution

Instances
  ExperimentControl : ThreadedMixedComp (1,0,0,0,0,0,0,2)
  DoctrineAuthoring : DynamicServerized(1,3)
  DoctrineValidation : DoctrineValidationT
  TrackServer : ThreadedMixedComp (3,0,1,0,1,0,0,0)
  GeoServer : GeoServerT
  DoctrineReasoning : DoctrineReasoningT
  DisplayServer : DisplayServerT
  CS$_{1..4}$: ClientServer
  DCS$_{1..5}$: DClientServer
  DCSPush$_{1..4}$: DClientServerPush
  OpenLoop : OpenLoopBuffer
Attachments
  ExperimentControl.DClientPush **as** DCSPush$_1$.Client
  DoctrineAuthoring.Service **as** DCSPush$_1$.Server
  ExperimentControl.DClientPush **as** DCSPush$_2$.Client
  DoctrineValidation.ExCtrl **as** DCSPush$_2$.Server
  ExperimentControl.ServerPush **as** CS$_1$.Server
  TrackServer.ClientPull **as** CS$_1$.Client
  DoctrineAuthoring.Client **as** DCS$_1$.Server
  DoctrineValidation.DoctAuth **as** DCS$_1$.Client
  TrackServer.ServerPush **as** DCS$_2$.Server
  DoctrineValidation.TrSrv **as** DCS$_2$.Client
  DoctrineAuthoring.Client **as** DCS$_3$.Server
  DoctrineReasoning.DoctAuth **as** DCS$_3$.Client
  DoctineAuthoring.Client **as** DCS$_4$.Server
  GeoServer.DoctAuth **as** DCS$_4$.Client
  TrackServer.ServerPush **as** CS$_2$.Server
  DoctrineReasoning.TrSrv **as** CS$_2$.Client
  TrackServer.ServerPush **as** CS$_3$.Server
  GeoServer.TrSrv **as** CS$_3$.Client
  GeoServer.DoctReas **as** OpenLoop.Source
  DoctrineReasoning.GeoSrv **as** OpenLoop.Target
  DoctrineAuthoring.Client **as** DCS$_5$.Server
  DisplayServer.DoctAuth **as** DCS$_5$.Client
  TrackServer.ServerPush **as** CS$_4$.Server
  DisplayServer.TrSrv **as** CS$_4$.Client
  DisplayServer.DoctVal **as** DCSPush$_3$.Server

DoctriveValidation.DispSrv **as** DCSPush$_3$.Client
DisplayServer.DoctReas **as** DCSPush$_4$.Server
DoctrineReasoning.DispSrv **as** DCSPush$_4$.Client
end Testbed2.

8 Evaluation

We have seen that WRIGHT can be used to provide a formal specification of the style used by the AEGIS prototype implementors. As we have illustrated, this specification illuminates many of the issues left unresolved in a less formal treatment.

The primary benefit of this specification has been its precision and its attention to detail. We note, however, that although we have been quite specific about the protocols of interaction, the specification has abstracted considerably from the actual functional behavior of the components in the system.

A secondary benefit of the treatment that we have given is the ability to reason about the architectural style within which AEGIS was developed. Although space did not permit it here, arguments about absence of deadlock and about substitutability of one connector type for another can be made in a rigorous fashion. These results become general rules that can be applied to all instances of the style. Hence the architectural level specification becomes cost effective through amortization of its results across a wide variety of systems.

Although the specifications shown in this paper are complex, they provide a basis for precise reasoning about the system and may be used, in other instances of this same style, as a basis for more concise descriptions of functioning systems. In effect, they act to give meaning to the informal descriptions such as shown in figure 1, and as building blocks for a family of precise architectural specifications.

On the negative side, this specification highlights some of the weaknesses of WRIGHT. The most glaring is the fact that the process structure of a WRIGHT description is (like CSP) static: new components and connections cannot be created on the fly. We were able to circumvent the problems in the case of the dynamic connectors, by simulating the opening and closing of a connection, and also by the use of parameterized components. But for an architecture with more dynamic behavior – e.g., new components are created while the system is running – this kind of model would not suffice. However, in exchange for this limitation, we get all of the benefits of CSP: its algebraic rules, its simple treatment of refinement, and its capability for reasoning about deadlock.

WRIGHT is not, however, intended to stand alone as the only architectural representation of a fully functioning system. Other formalisms will be more appropriate for the exploration of issues such as run-time performance, user- and machine-interfaces, allocation to hardware, and configuration maintenance. The software architect must confront many issues in developing the system, and so we would expect many tools to be deployed.

In addition, the structuring of WRIGHT is not uniquely tied to the CSP mechanisms described in this paper. We are exploring, in a different case study, ways that we can use WRIGHT-like descriptions with Z as the semantic base to provide a data-oriented view of a system's architecture.

In the broader scheme of things, it will be interesting to compare our specification with others of the same system. Through such comparisons we can begin to understand what are the tradeoffs in expressiveness and reasoning capability of alternative formal models.

References

[AAG93] G. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proc. of SIGSOFT'93: Foundations of Software Eng.*, Software Eng. Notes 18(5). ACM Press, Dec 1993.

[AG92] R. Allen and D. Garlan. A formal approach to software architectures. In Jan van Leeuwen, editor, *Proc. of IFIP'92*. Elsevier Science Publishers B.V., Sept 1992.

[AG94a] R. Allen and D. Garlan. Formal connectors. Technical Report CMU-CS-94-115, Carnegie Mellon University, Mar 1994.

[AG94b] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. of the 16th International Conf on Software Eng.*, May 1994.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, (96), 1992.

[GN91] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*. Springer-Verlag, LNCS 551, Oct 1991.

[GPT95] David Garlan, Frances Newberry Paulisch, and Walter F. Tichy, editors. *Summary of the Dagstuhl Workshop on Software Architecture*, Feb 1995. Reprinted in ACM Software Eng. Notes, July 1995.

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

[IW95] P. Inverardi and A. Wolf. Formal specification and analysis of software architectures using the chemical, abstract machine model. *IEEE Trans. on Software Eng.*, 21(4), Apr 1995.

[L+95] D. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE Trans. on Software Eng.*, 21(4), Apr 1995.

[MK95] J. Magee and J. Kramer. Modelling distributed software architectures. In *Proc. of the 1st International Workshop on Architectures for Software Systems*. Carnegie Mellon University Technical Report CMU-CS-95-151, Apr 1995.

[MQR95] M. Moriconi, X. Qian, and R. Riemenschneider. Correct architecture refinement. *IEEE Trans. on Software Eng.*, 21(4), Apr 1995.

[S+94] M. Shaw et al. Candidate model problems in software architecture. Draft Publication, 1994.

[YS94] D. M. Yellin and R. E. Strom. Interfaces, protocols, and the semi-automatic construction of software adaptors. *Proc. of OOPSLA'94*, Oct 1994.