

Connectors in Configuration Programming Languages: are They Necessary?

Judy Bishop and Roberto Faria
Computer Science Department
University of Pretoria
Pretoria 0002, South Africa
jbishop, roberto @cs.up.ac.za

Abstract

Configuration programming is the process whereby components written in any conventional programming language can be bound together to form a dynamic system, often suitable for execution on distributed hardware. Among the specialised languages that exist for configuration programming there is currently a debate over the importance of recognising the connections between components as being as important as the components themselves. This paper lays out the pros and cons of the debate, outlining in the process the properties and roles of connectors. By means of experiments we show how connectors influence the way configurations are programmed and also how some of the effects can be simulated. The examples are given in Darwin, UNICON and WRIGHT and reference is also made to the status of other current configuration languages.

Introduction

Software architecture is an emerging discipline which aims to enable system designers to express the style of a design in such a way that it can be recognised as a pattern later, and reused in an appropriate, similar context. A fundamental component of software architecture is therefore the expression of these styles and patterns. While work is going on at the higher level of pattern design – often under the title of **frameworks** – a considerable body of practical knowledge and tool support has been built up at a different level, that of **configuration programming** (CP) [Kramer 1990, Bishop 1994, Shaw 1995].

In configuration programming, the system designers separate the **computation** involved in achieving the purpose of the system from the **connection** between the modules that implement that computation. The idea is that:

- a) the connection might change as the system evolves;
- b) the computation modules could be reused in other systems with different connections.

The computation is expressed in ordinary programming languages – C++, Ada, Pascal, even FORTRAN – while the connection is the responsibility of a specialised language.

The purpose of the **configuration language** (CL) is to provide the expressive power for the “glue” needed to put components together in a manner that is **understandable, adjustable, secure** and **efficient**.

Schwanke [1994] has echoed the needs of the industrial community in saying that “the most pressing architectural concern is maintaining consistency between the architecture and the code” over periods extending for upwards of fifteen years. Understanding the architecture is therefore important, but so is the need to be able to adjust its structure.

New components which can replace existing ones may emerge as time goes by, and new ways of connecting them may be discovered. Configuration programming considers both legacy systems comprising old components as well as future systems. In both cases, the behaviour of the components may not be immediately evident and the CL can provide the flexibility needed to incorporate them gradually.

The third attribute, security, implies that the configuration process should enable errors in design and implementation to be detected before the system is executed. In the first instance, a CL will check that the communication between components agrees in the type of data and the method of its transmission. A CL can also provide mechanisms for dynamic configuration, which can support the reliability requirements of safety critical systems running on distributed hardware.

Lastly, dynamic and distributed systems often need to respond to events in real-time, and therefore the performance of the final system must be assured. Fortunately, it has been shown in a range of studies [Purtilo 1993, Magee 1994, Shaw 1995] that efficiency is not compromised if systems are built at two levels – computation and configuration – and then compiled using a CL compiler.

Development of configuration languages

Since the 1970s when they were first mooted, languages for CP have been through several stages of development, as summarised in Figure 1. The classification of a language as a PL, MIL, IDL, CPL or ADL is based on increasing power. Although useful for the purposes of this table, it must be stressed that these terms are used very loosely in the literature and there is no general agreement as to which label a language should be given. A more detailed comparison of CLs is given in [Bishop 1994].

Referring to the table, we see that the first languages that come into consideration are the modular programming languages of the 1970s and 1980s. Modules can define interfaces consisting of data and routines, to be accessed by other modules. The type checking of the correct use of the interfaces is done by the compilers or binders.

The issue of component types is crucial in CP, and was ducked in the original Ada (although some components could be generic). The idea is simple: every component in the system is instantiated from a component

type, even if the type is only used once. Component types can in most CLs be grouped in hierarchies so that higher level abstractions can be formed. The advantage of the combination of types and hierarchies is that reuse of configurations becomes possible, as shown in the next column of the table.

The last column looks at the kinds of relationships or connections that can exist between components. In PLs, these can be anything that the language supports. For example, Ada packages can communicate by means of procedure calls and shared data, and Ada tasks by means of the rendezvous. The specialised CLs settled on one means of communication between components. For example, PCL defines a generic channel package called DATA_OBJECT which can be instantiated for different types of messages, and from which components call appropriate procedures, such as open, read and write. The important point is that the semantics of communication is defined by the CL and known by the components.

The middle group of languages exhibit many of the properties of MILs and some of those of the next group.

	Sample languages	Separate language for system design	Type-checking between components	Hierarchies of component types	Reuse of a system patterns	Variety of connections between components
programming language (PL)	e.g. CLU, MESA, Ada, C++	no	static	not always	no	some e.g. procedure call, shared data, rendezvous
module inter-connection language (MIL)	e.g. MIL75 etc. see [Prieto 1986] PCL [Dobbing 1993]	yes	static	not necessarily	no	usually procedure call and shared data
interface definition language (IDL)	Durra [Barbacci 1993], Polyolith [Purtilo 1994]	yes	static or dynamic	sometimes	sometimes	usually only one e.g. rendezvous or RPC
configuration programming language (CPL)	Darwin [Magee 1994]	yes	static or dynamic	yes	yes	mostly uses supporting environment but extensible
architectural description language (ADL)	UNICON [Shaw 1995], WRIGHT [Allen 1994a]	yes	static or dynamic	yes	yes	explicit selection, and also possibly user defined

FIGURE 1 Stages in the development of configuration programming languages

Durra, which was defined at the SEI for fault tolerant applications in Ada, specifies communication through components which are fixed types called channels, although the actual implementation of a channel package can be modified to take account of different properties. Polyolith from the University of Maryland connects components to a Polygen software bus which implements the connection mechanism. Different applications can select different buses, and it is also possible that an application can migrate to a new bus.

In the last two categories of CLs, more flexibility is evident. Darwin comes from a line of well established CLs developed at Imperial College and makes use of a selection of communication mechanisms supplied by its underlying support environment (The most well-developed of these is Regis). Typically these are ports, events, streams and so on. UNICON and WRIGHT emerged from different groups at Carnegie Mellon University and both have the explicit goal of providing the user with the ability to define new connection mechanisms for a particular configuration.

The issue that this paper addresses then is: are the goals of understandability, adaptability and security better attained if the connection between components is given prominence in the language, or are connectors unnecessary baggage that can be easily handled by defaults in the language or by well-defined components? We examine the question by first looking at the fundamentals of connection and binding. Then we consider how connectors are realised in two current languages – UniCon and Wright. Thereafter we present the case of a language without connectors – Darwin. This leads to a set of connector properties and the conclusions.

Approaches to connectors

The proponents of connectors do not envisage them as simply another form of component. Allen [1994a] defines connectors as “protocols that capture the expected patterns of communication between modules”. Shaw [1994] states: “A connector mediates the interaction of two or more components. It is not in general implemented as a single unit of code to be composed.”

In order to perform its mediating role, a connector needs a specification of the **type** of connection it provides, and the **roles** that need to be played by the components it connects. In general a connector is not binary (i.e. between two components) but may be Nary (between several components). Three approaches to connectors can be identified, in increasing order of sophistication:

1. Implicit connection
2. An enumerated set of built-in connectors
3. User-defined connectors.

Suppose we have two components A and B, and we bind them together in a hypothetical configuration language with implicit connection based on the message-passing paradigm as follows:

```
instantiations
  A : A_component_type;
  B : B_component_type;
bindings
  A.output - - B.input;
```

The binding here is between the two **ports** output and input which must have the same **signature** (data type). The binding permits component A to call port i/o procedures which will transmit data to corresponding procedures in B. Within the code of the components (written in a programming language) we would thus have

```
output.send(length * height); - - in component A

input.receive (area); - - in component B
```

The connection is tightly coupled and the component has to be aware of the port facility since it must call the procedures. Since ports will be implemented in some underlying programming language (such as C++) it is possible that variations may be included, for example to provide dynamic name binding through port references (as in Darwin [Magee 1994]), but the fundamental nature of the connection does not change.

Consider now a set of built-in visible connectors. The example would be extended to include:

```
instantiations
  C : A_B_connector;
bindings
  A.output - - C.requires;
  C.provides - - B.input;
```

The connection represented by C can be any of a variety of mechanisms such as a pipe, a rendezvous or a remote procedure call. The behaviour of A and B will be checked to see that it conforms to that which is expected by C. So, for example, if A_B_connector has Unix pipe semantics (a very common connector choice) then the C *requires* and *provides* roles will be a source and a sink, and A and B must have players to fulfill these roles. (The analogy of a stage play is a sound one: the connector is the script, the roles are listed as available, and players must step forward to play the roles correctly according to the script.)

In this instance, the connector type and the roles played by the component are visible in the configuration specification. Therefore if A_B_connector was some other connector type, such as a remote procedure call, the roles for C will be something like a definer and a caller, and there would exist in A and B the appropriate statements in the programming language, for example:

```
remote procedure foo (x : integer); -- in A
call foo (4); -- in B
```

In the third approach, we could define our own protocol for the connection. It may also be possible to experiment with the connection while leaving the components unchanged.

We now consider configuration languages which have the latter two approaches, – UniCon and WRIGHT, and follow that with a look at an approach 1 language – Darwin.

Built-in connectors in UNICON

Specifying the interaction between components is not simple, and two attempts have been made at different levels. The UNICON language [Shaw 1995] gives a choice of seven **built-in connector** types – pipe, file, procedure call, remote procedure call, PLBundler, data access and real time scheduler. Each of these defines a set of role types that must link up to player types of specified component types which are valid in the given context. These are shown in Figure 2.

From Figure 2, it can be seen that UNICON includes eight different component types. These are necessary so that the checking against the usage of particular connectors can proceed. The list of allowable components for each connector is intended to be as all-embracing as possible.

The semantics of UNICON's built-in connectors are defined as part of the language, and are intended to correspond to the usual interactions supported by operating systems and languages. In order to provide for a richness in the abstraction, there is some overloading of the abstractions. For example, the Unix pipe mechanism is represented by two connectors: Pipe and FileIo. Pipe may link filters or sequential files, or combinations of these,

whereas the FileIo connector is intended for modules or sequential files. PLBundler is an abstraction for connecting a collection of procedure or data definitions with their calls and uses, and is included in the list in order to reduce the number of individual connections that would need to be made in a large system.

Consider an example of using this rich language. A procedure call connection could be established between the roles of a Definer and a Caller, defined in the protocol of a connector's interface. In components of type Computation or Module there would be players of the types RoutineDef and RoutineCall which could then bind to an instance of Definer and Caller. These associations are shown diagrammatically in Figure 3.

Example

The example shows system with a single pipe connector linking two components. CatFile collects data and RemoveVowels edits it. The three items are grouped together into a higher level component with access to the standard Unix input-output environment, as shown in Figure 4.

The UNICON code for setting up such a system is given in Figure 5. System is a component which is defined to be of type Filter. We can define as many players as we like in its interface, as long as they are of the two prescribed types for a Filter: StreamIn and StreamOut. In this case we have three players: input, output and error.

CatFile and RemoveVowels are also Filters with similar interfaces to System's, except that the port bindings which System uses to link to the underlying environment are omitted. Instead, the "glue" is provided in System's implementation part. The implementations of these two components are actual executable files originating in a programming language, and stored in the file system as indicated in their implementation parts.

Connector Type	Role types	Component Types	Player Type
Pipe	Source and Sink	Filter	StreamOut and StreamIn
		SeqFile	ReadNext and WriteNext
FileIO	Reader and Writer	Module	ReadFile and WriteFile
	Readee and Writee	SeqFile	ReadNext and WriteNext
ProcedureCall	Definer and Caller	Computation or Module	RoutineDef and RoutineCall
RemoteProcCall	Definer and Caller	Process or SchedProcess	RPCDef and RPCCall
PLBundler	Participant	Computation, Module or SharedData	PLBundle, RoutineDef, RoutineCall, GlobalDataUse, GlobalDataDef
DataAccess	Definer and User	Shared Data or Module, plus Computation (for use)	GlobalDataDef and GlobalDataUse
RTScheduler	Load	SchedProcess	RTLLoad

Figure 2. The built-in connectors of UNICON (adapted from Shaw 1995]

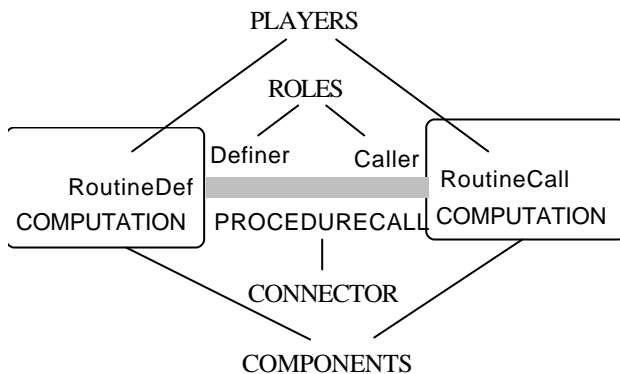


Figure 3 Relationship between items in UNICON

The connector follows the same pattern in that it selects the appropriate role types for a Pipe, namely sink and source, and creates three roles using these types. MAXCONNS is an attribute which applies to a role and indicates that only one binding can be made. Thus connectors can be varied in terms of the number of players they provide, as well as the specific attributes of these roles. Other attributes for connector roles are MINCONNS and ACCEPT, which can restrict the number and type of players that can serve in this role.

In UNICON there is also provision for high-level intentions of time, reliability, ordering, performance and so on to be specified through the connectors, and checked.

User-defined connectors in WRIGHT

The language which has gone the furthest towards supporting user-specified connectors is WRIGHT [Allen 1994a]. WRIGHT has a rich semantics for specifying the

behaviour of connectors, based on Hoare's CSP. Figure 6 shows how the simple example would be defined in WRIGHT. Information about WRIGHT is not as completely codified yet as for the other two languages, thus the example is not quite complete. Reference to the error ports is omitted, and the hierarchy of System and the other components is flattened.

The connector specifies that there are two role players, one that gets and one that puts. The glue code then regulates this process, ensuring that the eof detected on the source is correctly conveyed to the sink, and causes a shut down. The Filter component type is specified similarly, but the difference between the behaviours of the two is evident. The Filter promises to consume all the data presented to it at the port input. The source port in the Pipe, which will be connected to input, does not make such a promise, but may choose to close down. The symbol ? indicates a non-deterministic choice, made by the process itself. o on the other hand is a deterministic choice made by the environment.

The modified CSP used in WRIGHT enables sophisticated protocols to be specified, and properties about them to be proved [Allen 1994b].

Emulating connectors in Darwin

As with UNICON, developing a system in Darwin involves constructing systems from simple components and connections between them. Components are strongly typed first class language primitives in Darwin and composite components can be formed from simpler ones. As well as allowing for nested structuring, Darwin supports incremental structuring by extension.

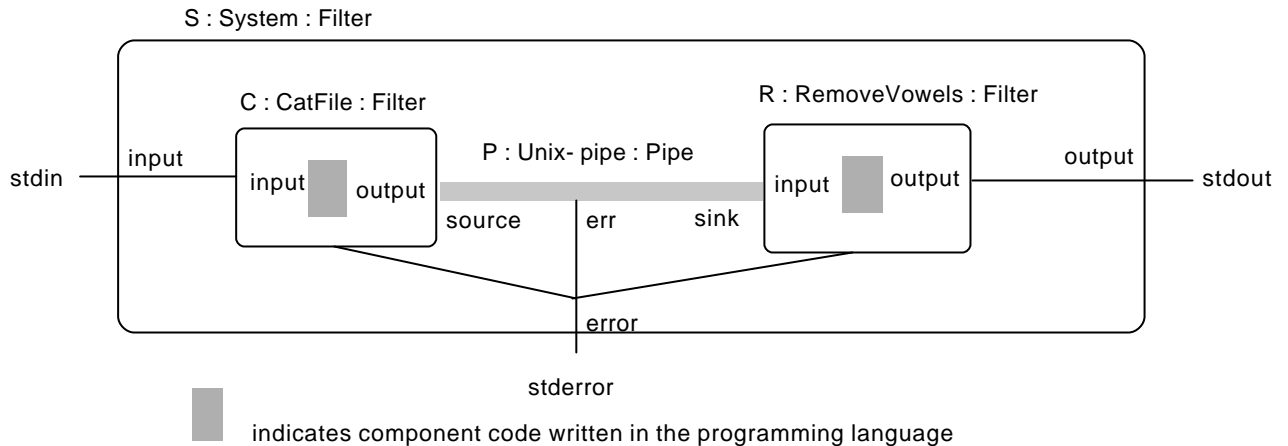


Figure 4 A Filtering System Example

```

COMPONENT System
INTERFACE is
  TYPE Filter
  PLAYER input IS StreamIn
    SIGNATURE ("line")
    PORTBINDING (stdin)
  END input
  PLAYER output IS StreamOut
    SIGNATURE ("line")
    PORTBINDING (stdout)
  END output
  PLAYER error IS StreamOut
    SIGNATURE ("line")
    PORTBINDING (stderr)
  END error
END INTERFACE
IMPLEMENTATION IS
  USES C INTERFACE CatFile
  USES R INTERFACE RemoveVowels
  USES P PROTOCOL Unix-pipe
  BIND input TO C.input
  BIND output TO R.output
  BIND C.error TO error
  BIND R.error TO error
  BIND P.err to error
  CONNECT C.output TO P.source
  CONNECT P.sink to R.input
END IMPLEMENTATION
END System

INTERFACE is
  TYPE Filter - - as before without portbindings
END INTERFACE
IMPLEMENTATION IS
  VARIANT RemoveVowels IN "remove"
    IMPLTYPE (Executable)
  END RemoveVowels
END IMPLEMENTATION

CONNECTOR Unix-pipe
PROTOCOL IS
  TYPE Pipe
  ROLE source IS source
    MAXCONN (1)
  END source
  ROLE sink IS sink
    MAXCONN (1)
  END sink
  ROLE err IS sink
    MAXCONN (1)
  END err
END PROTOCOL
IMPLEMENTATION IS
  BUILTIN
END IMPLEMENTATION
END Unix-Pipe

```

Figure 5. UniCon code for a simple System

```

COMPONENT CatFile
INTERFACE is
  TYPE Filter - - as before without portbindings
END INTERFACE
IMPLEMENTATION IS
  VARIANT CatFile IN "catfile"
    IMPLTYPE (Executable)
  END CatFile
END IMPLEMENTATION
END CatFile

COMPONENT RemoveVowels

```

```

System Example
Component Filter
port input = get ? x -> input o get-eof -> close -> ?
port output = put ! x -> output ? !close -> ?
comp spec =
  let Close = Input.close -> output.close -> ?
  in Close o input.get?x -> output.put!x
  -> (close o computation)

Connector Pipe =
role source = put ! x -> source ? close -> ?

```

```

role sink =
  let Exit = close ?
  in let DoGet = (get -> sink o eof -> Exit)
  in DoRead?? ?Exit
glue = let GetOnly = sink.get -> GetOnly
      o sink.eof -> sink.close ->?
      o sink.close ->?
  in let PutOnly = source.put -> PutOnly
      o source.close ->?
  in sink.put -> glue
  o source.get -> glue
  o sink.close -> GetOnly
  o source.close -> PutOnly

Instances
  S : Filter
  C : Filter
  R : Filter
  P : Pipe

Attachments
  S.input as C.input
  C.output as P.source
  P.sink as R.input
  R.output as S.output
end Example

```

*Figure 6. The example in Wright
(excluding treatment of the error ports)*

This feature is provided by allowing component types to be declared as derived component types (similar to single level inheritance in object-oriented languages).

While Darwin allows the programmer to specify which components are to be connected together, it does not consider connections as first class language primitives. It has always been Darwin's standpoint that the component concept is powerful enough to encompass the effect of connectors as defined in, say, UNICON .

Example

Consider the simple example posed in Figure 4. In a normal Darwin system, CatFile and RemoveVowels would be bound together directly. The components themselves would have to include calls to in and out methods defined for a **port** class [Magee 1994].

The C++ for the components would be expressed as constructors which are linked in with header files produced as a result of the corresponding Darwin components. It is in these components that the ports will be defined. In this example, Line is a class defined for the type of data being passed between the components.

```

component Environment
{
  inst
  S : System;

```

```

  stdin: StandardInput;
  stdout: StandardOutput;
  stderr: StandardError;

bind
  stdin.yield -- S.input;
  S.out put -- stdout.receive;
  S.error -- stderr.receive;
}

component Filter
{
  provide input <port line>;
  require output <port line>;
  require error<port line>;
}

component Pipe
{
  provide source <port line>;
  require sink <port line>;
  require error <port line>;
  bind source -- sink;
}

component RemoveVowels: Filter;
component CatFile: Filter;
component System: Filter
{
  inst
  C : CatFile;
  R : RemoveVowels;
  P : Pipe;
  bind
  input -- C.input ;
  C.error -- error;
  C.output -- P.source;
  P.sink -- R.input;
  R.output -- output;
  R.error -- error;
}

component StandardError
{
  provide receive<port line>;
}

component StandardOutput
{
  provide receive<port line>;
}

component StandardInput

```

```

{
  require yield<port line>;
}

```

Figure 7 The simple example in Darwin

It serves the same purpose as the SIGNATURE("line") statement in the UNICON version (Figure 5). The system can then run with an underlying support environment such as Regis.

Now one could insert a pipe component between CatFile and RemoveVowels, and the Darwin program would be as in Figure 7. No implementation is required for Pipe since all it does is connect two already existing computations. The loops and checking for end of data and so on are all embodied in the C++ components. So what does it buy us? Well, having a defined connector between the two components raises the level of abstraction, so that the reader can see and be assured that the communication is pipe-like, not via RPC, say.

However, the assurance is flimsy, because it is based only on the association of the **name** of the connector with an existing well-known mechanism. If we had called the connector XYZ, then a reader would have been none the wiser, without delving into the C++ code. Similarly, if we had left the connector as pipe, and then changed the code in CatFile and RemoveVowels to reflect something different, such a discrepancy could not be detected.

Pipe is a reasonably easy abstraction to insert into Darwin programs. However, the other connectors offered by UNICON or modelled in WRIGHT would have to be modelled laboriously.

Properties of connectors

We can now arrive at the following list of properties for connectors in general:

1. Connectors provide a means for checking the types of communication (signatures) between components in a system.
2. They enable allowable roles to be defined for components that are to be connected, and for their mutual behaviour (in terms of protocols) to be checked at compile time.
3. The allowable roles may be just one (a built-in mechanism), a fixed number (enumerated connectors) or unlimited (user defined definitions).
4. Available connectors should include those commonly supported by the operating system and the programming language, such as pipes and the

RPC on the one hand, or shared data and the rendezvous on the other.

5. Connectors can enable high-level intentions of time, reliability, ordering, performance etc. to be specified and checked.

Further work

The existing languages for CP are referred to in Figure 1. Of recent times, connectors have become topical and a language bred in industry, Gestalt [Schwanke 1994], is based heavily on the ideas of both UNICON and WRIGHT. Work has also proceeded on the theoretical side, with a taxonomy of system structures and a notation for their connections being described in [Dean 1995]. Proofs of correctness of connectors can be attempted with WRIGHT's precise notation [Allen 1994b].

Mention should be made of the recent extensions by the UniCon team [Shaw 1996] which gives more precise definitions of the connectors, and explains how the UniCon compiler realises connectors from available intermediate products using information localised into **experts**. Further work in the creation of **styles** for architectural modelling has been done by [Allen 1995].

Conclusions

This paper has described what connectors look like and shown a few variations of them. It would be easy to dismiss connectors as just a "nice to have but hard to implement" feature of configuration programming. Are they really necessary?

The question must be answered with software architecture in mind. The fact that there are different communication mechanisms in existence, such as RPC, shared data or message passing, and that components, both existing and future cannot be constrained as to which they will use, indicates that a configuration language needs to be flexible enough to accommodate this variety.

Since Darwin does not have connectors, it relies on underlying implementations, for example C++ classes defined for ports in Regis. The fact that the coupling is loose is both an advantage and a disadvantage: components can be relinked, which is good, but there is also a loss of information when the connector information is actually coded down at the component level.

UNICON on the other hand scores because it has carefully crafted a set of likely connectors and has arranged for these to be directly linked into the existing

constructs in the operating system or language. Although WRIGHT may seem to be at the top of there evolutionary tree, there remains a problem in that the formal specification of a protocol has still to be correctly realised in code.

Ultimately, it is the goals of understandability and adjustability that must be met, and making connectors visible and more checkable achieves these more so than hiding them in a layer below the CL. The conclusion of this paper is therefore that yes, they are necessary for secure, successful configuration programming in the future.

Acknowledgments

The Foundation for Research Development provided support during the period of this work. We would like to thank the referees for their insightful questions and comments.

References

- [Allen 1994a] Allen R and Garland D, *Beyond definition/use: Architectural interconnection*, Proc. Workshop on Interface Definition Languages, Portland, Oregon, Jan. 1994, in Sigplan Notices **29** (8) 35–45, August 1994.
- [Allen 1994b] Allen R and Garland D, *Formalizing architectural interconnection*, Proc. 16th Int'l Conference on Software Engineering (ICSE), Sorrento, Italy, May 1994.
- [Allen 1995] Allen R and Garland D, *A case study in architectural modelling: the AEGIS system*, unpublished manuscript, Carnegie-Mellon University, July 1995.
- [Barbacci 1993] Barbacci M R *et al*, *Durra: a structure description language for developing distributed applications*, Soft. Eng. J., **8** (2) 83–94, March 1993.
- [Bishop 1994] Bishop J M, *Languages for configuration programming: a comparison*, IEEE Trans. Soft Eng. to appear, also UP CS Tech Report 94/04.
- [Callahan 1991] Callahan J R and Purtilo J M, *A packaging system for heterogeneous execution environments*, IEEE Trans. Soft. Eng. **17** (6) pp 626–635 June 1991
- [Dean 1995] Dean T and Cordy J R, *A syntactic theory of software architecture*, IEEE Trans on Soft Eng. **21** (4) 302–313, April 1995.
- [Dobbing 1993] Dobbing B, *Experiences with the partitions model*, Ada Letters, XIII (2) pp 65–77, March/April 1993
- [Kramer 1990] Kramer J, *Configuration Programming – a framework for the development of distributable systems*, Proceedings of the IEEE Int'l Conference on Computer Systems and Software Engineering (CompEuro 90), Israel, May 1990.
- [Magee 1994] Magee J, Dulay N and Kramer J, *Regis: a constructive development environment for distributed programs*, Distributed Systems Engineering Journal **1** (5) 304–312 September 1994.
- [Prieto 1986] Prieto-Diaz R and Neighbours J M, *Module interconnection languages*, Journal of Systems and software **6** 307–334, 1986
- [Purtilo 1993] Purtilo J, *The Polyolith software bus*, ACM Trans. on Prog. Lang. and Sys., **16** (1) 151–174 January 1994
- [Schwanke 1994] Schwanke R W, Strack V A, Werthmann-Auzinger T, *Industrial software architecture with Gestalt*, Siemens Technical Report, Princeton NJ, 1994.
- [Shaw 1994] Shaw M, *Procedure calls are the assembly language of software interconnection: connectors deserve first-class status*, CMU-SEI Tech Report 94-TR-2
- [Shaw 1995] Shaw M, DeLine R, Klein D V, Ross T L, Young D M and Zelesnik G, *Abstractions for software architecture and tools to support them*, IEEE Trans. Soft. Eng. **21** (4) 314–335, April 1995
- [Shaw 1996] Shaw M, DeLine R and Zelesnik G, *Abstractions and Implementations for architectural connections*, 3rd

International Conference on Distributed Computing
Systems, Annapolis, May 1996