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.

Keywords: configuration programming, connectors, MILs, software architecture, UniCon, Darwin, WRIGHT.

1. 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** [Kramer 1990, Bishop 1994, Shaw 1995b].

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 the

a) the connection might change as the system evolves (during development or dynamic reconfiguration), and

b) the computation modules could be reused in other systems with different connections.

The computation is expressed in ordinary programming languages – C++, Ada, Pascal – while the connection is the responsibility of a specialised language. Unfortunately, there is no agreement among the community as to the term to be applied to these languages as a group, with the following in popular use: module interconnection language (MIL), interface definition language (IDL), software architectural description language (SARL) and configuration language (CL). We shall use the latter in this paper, since MIL is a somewhat dated term, IDL has been over-used as an actual product name, and SARL is a bit futuristic for the level at which we are dealing.

The purpose of the **configuration language** is to provide the expressive power for the "glue" needed to put components together in a manner that is

- understandable,
- 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. Understandability is therefore an important consideration. Since many of the systems that need dynamic configuration or that will run on distributed hardware are safety critical (transport systems, for example), security is also an issue.

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 at least two studies [Magee 1994a, Shaw 1995b] that efficiency is not compromised if systems are built at two levels and compiled using configuration language compilers, as shown in Figure 1.

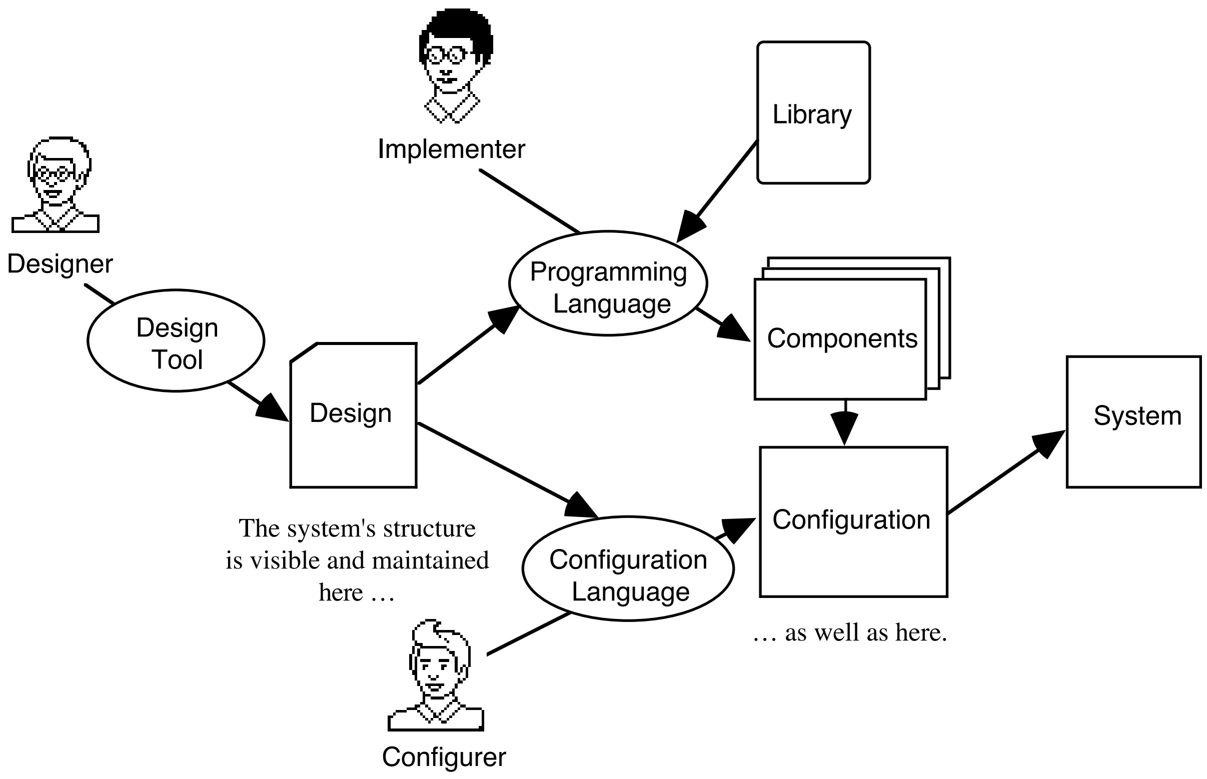


Figure 1 The role of a configuration language

At present, some ten configuration languages can be identified, of which about half are in practical use. While each addresses specific concerns and is a product of the era in which it was first designed, there is considerable commonality among the features they offer. Figure 2 (extended from [Bishop 1994]) shows the range for some current languages [Magee 1994a], [Barbacci 1993], [Dobbing 1993], [Callahan 1991], [Shaw 1995], [Allen 1994]..

	Darwin	**Durra**	**PCL**	**Polylith**	**UniCon**	**WRIGHT**
process term	process	component = task or channel	partition = RCI designated library unit	module	language dependent	language dependent
process language	C++ plus library template classes	Ada83 plus generic package	Ada83 or Ada9X plus RCI pragma and generic package	any + Polylith primitives	any	any
connections	require and use ports	channels package with ports	data_object package with channels	define and use an interface	ports and a set of connector types	user-defined connectors in WRIGHT
component term	component	compound component = task	program	application	component	component
component hierarchy	yes	yes	no	no	yes	yes
binding	explicit and includes conditionals	explicit via channels	explicit and includes rejoining	explicit or implicit	explicit to ports and connectors	explicit
communi-cation	synchronous and asynchronous message passing via a network	synchronous message passing in multi-threaded clusters	synchronous message passing point-to-point via an occam harness	synchronous message passing on a software network bus	complete variety from pipe to RPC to shared data access	anything that can be defined in the language
language environment	C++ with Regis on top of UNIX	cluster manager on DEC VAX	AdaMap on top of Alsys transputer Ada	Polylith, Polygen, Surgeon and Catalyst on DEC stations	Unix based with Odin and Mach (for RT Scheduler connectors)	not yet specified
software-hardware mapping	implicit but can be overridden	implicit	explicit	implicit with constraints	not yet specified	not yet specified
reconfigu-ration	no	yes via signals	no	yes in Surgeon and Catalyst	no	no

FIGURE 2. Comparison of optimal criteria and four languages

However, a question has arisen as to the need for **connectors** in the languages. Connectors are not components as such, but can be thought of as “defining a set of roles that specific named entities of the components must play” [Shaw 1995b].

The issue that this paper addresses then is: are□the goals of understandability 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 properties of connectors in general. Then we consider how they are realised in two current languages – UniCon and WRIGHT. Thereafter we present the case of a language without connectors – Darwin. After looking at related work, the conclusions present a short comparison of the three languages and an assessment of future directions.

2. Properties of 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 [1994b] 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 N-ary (between several components).) Four approaches to connectors can be identified, in increasing order of sophistication:

Level	Approach to connectors	Example
0	Interchange formats	RTF for formatted text or PICT for line drawings
1	A built-in paradigms	message passing in Darwin
2	An enumerated set of built-in connectors	the seven common cases in UniCon
3	User-defined connectors	defined in modified CSP in WRIGHT

The use of text as a connecting mechanism is well known, with the standard “send it in ASCII or Postscript” being popular for distributing data and documents. Other formats such as RTF and PICT can be used in a single system composed of different programs, where a word processor might incorporate the output of a draw utility, for example. Although of practical value, we shall not consider this form of connection further.

We now move on to an example which successively introduces the main points of the other three approaches.

Example

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

In a configuration language:

```
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

In a programming language:

```
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 1994a]), 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:

In a configuration language:

```
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, there will be no reference to the connection in the components at the programming language level, since we are picking up a connection facility (Unix piping) at the operating system level. Thus the components A and B will exist, and be piped together via C, with the connection formed by the configuration.

Suppose instead that 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 caller, and there would exist in A and B the appropriate statements in the programming language, for example:

In a programming language:

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

call foo (4);   - - in B
```

After this introductory example, we can 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 infinite (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 should enable high-level intentions of time, reliability, ordering, performance etc. to be specified and checked.

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.

3. 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 1995b] 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 roles that must link up to player types of the component types which are valid in the given context. These are shown in Figure 3. All of them are reasonably self-explanatory except PLBundler: this is an abstraction for connecting a collection of procedure or data definitions with their calls and uses, and is presumably included in the list in order to reduce the number of individual connections that would need to be made in a large system.

For example, a procedure call connection could be established between a Definer and a Caller. Both of these would need to be components of type Computation or Module, and would need to provide players of the types RoutineDef and RoutineCall. Thus an instance of Definer would bind to an instance of RoutineDef and similarly for the calls.

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.

Connector Type	Roles	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	RTLoad

Figure 3. The built-in connectors of UniCon

Example 1

The first example shows a single pipe connector linking two components. These can be grouped together into a higher level component with access to the standard Unix input-output environment, as shown in Figure 4.

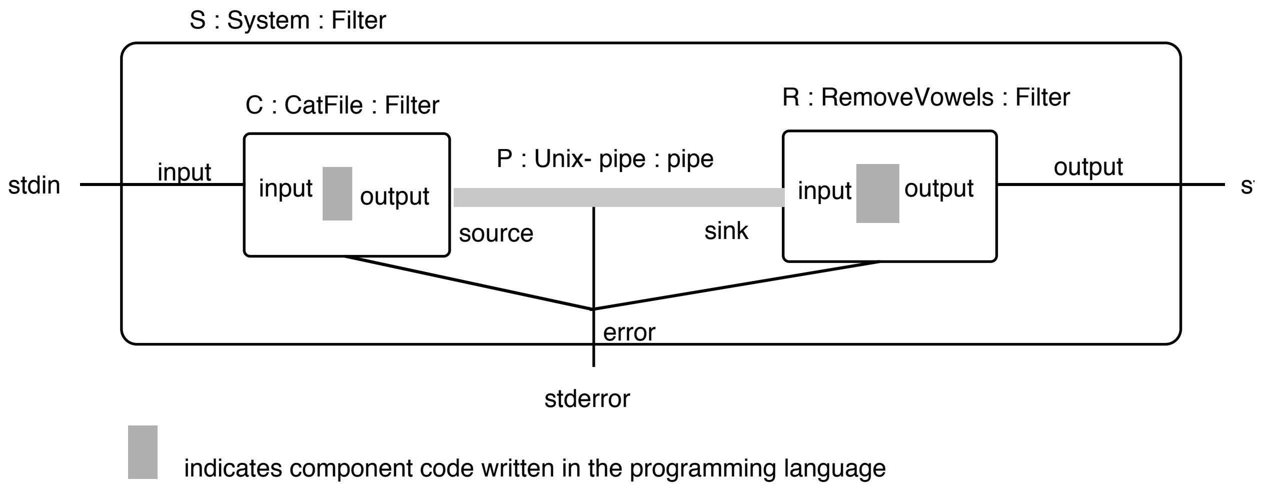


Figure 4 A Simple system in UniCon

The UniCon code for setting up such a system is given in Figure 5. Filter is a standard component type. CatFile and RemoveVowels are user defined components derived from this type. Their implementations are actual executable files originating in a programming language. Similarly, System has the same Filter interface, but has a more elaborate implementation consisting of the two filters and a pipe. Notice that UniCon supports two kinds of connection: port bindings, as well the roles of the names connectors.

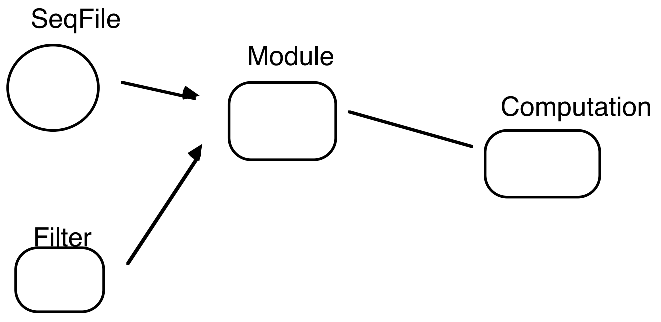
Example 2

Now consider a more complex example which illustrates the use of four connectors of three different kinds. The objective is to transfer data from a file and a filter on the left into a module, which is then called by a computation component on the right (6a). According to the table in Figure 3, computation components can connect via procedure calls or data access to other computations or modules. We shall choose a procedure call as the most appropriate connection here. The FileIo connector can bring data in from a sequential file to a module, and we have the first part of the problem solved (6b).

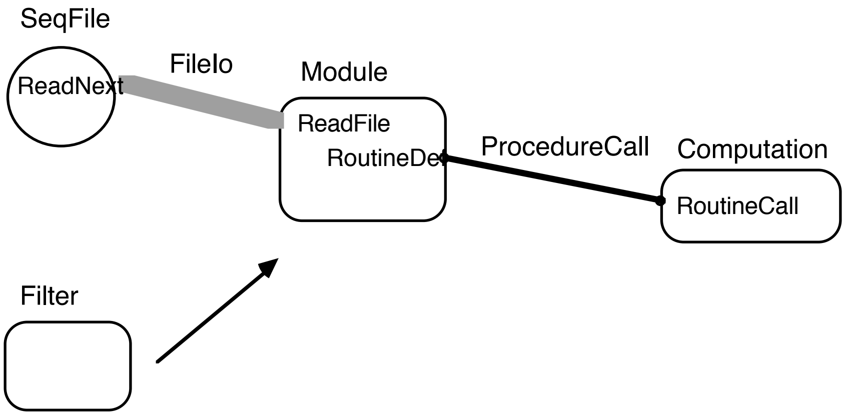
The second part is more complicated. A consequence of the UniCon approach of selecting a built-in set of connectors with strict rules about their use is that sometimes certain designs, such as this one, cannot be realised directly. Filters can only connect via pipes to other filters or to files. Filters cannot mix with modules, so we need to establish an intermediate file. The filter pipes to the file which can be correctly read by the module (6c).

```
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 ouput 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

COMPONENT CatFile
    INTERFACE is
        TYPE Filter - - exactly as before
    END INTERFACE
    IMPLEMENTATION IS
        VARIANT CatFile IN “catfile”
            IMPLTYPE (Executable)
            END CatFile
    END IMPLEMENTATION
END CatFile

COMPONENT RemoveVowels
    INTERFACE is
        TYPE Filter - - exactly as before
    END INTERFACE
    IMPLEMENTATION IS
        VARIANT RemoveVowels IN “remove”
            IMPLTYPE (Executable)
            END RemoveVowels
    END IMPLEMENTATION
END RemoveVowels

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

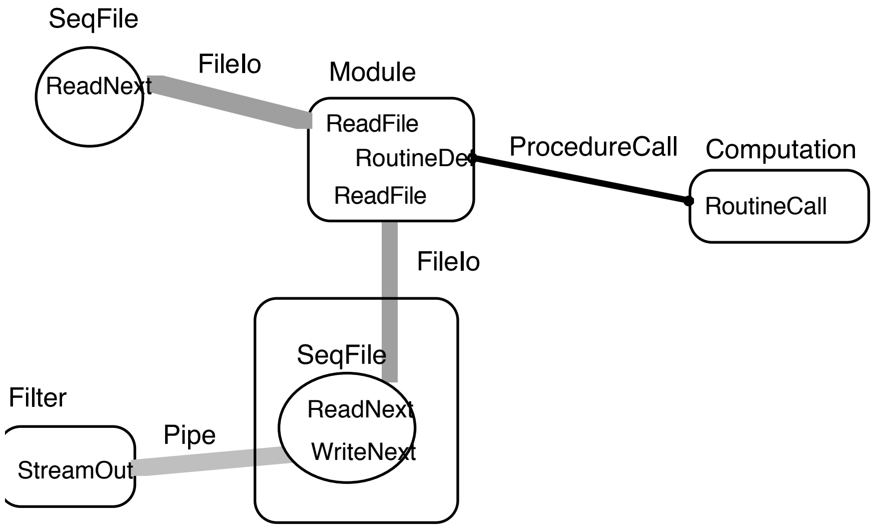
Figure 5. UniCon code for a simple System



6(a)



6(b)



6(c)

Figure 6. An example of connectors in UniCon

Thus the discipline of UniCon's connectors has ensured that we maintain the correct behaviour between components. We could not have connected the output of StreamOut to input via ReadFile using any of the built-in connectors. If a StreamOut to ReadFile connection is indeed a sensible thing to do, then the facility to be able to include **user-defined connector** types is necessary.

4. User-defined connectors in WRIGHT

The language which has gone the furthest towards supporting user-specified connectors is WRIGHT [Allen 1994b]. WRIGHT has a rich semantics based on Hoare's CSP for specifying the behaviour of connectors. Figure 7 shows how a simple connector for sharing data would be defined in WRIGHT.

```
connector Shared Data =
    role User1 = set -> User1 ∩ get -> User1 √
    role User2 = set -> User2 ∩ get -> User2 √
    glue = User1.set -> glue ❑ User2.set -> glue
        ❑ User1.get -> glue ❑ User2.get -> glue ❑√
```

Figure 7. A shared data connector in WRIGHT

The connector specifies that there are two role players for this connector. Each may get or set repeatedly. The glue code then regulates this process. In this simple version, there is no initialisation of the data, so a get may precede the first set. By redefining the roles to distinguish between an initiator and a user, and by expanding the glue description, this problem can be overcome in a variety of ways [Allen 1994b].

Example
Now consider how the example posed for UniCon might be implemented in WRIGHT (Figure 8). The important point is that FileIo connector can be so defined as to provide a service between files and the module or between filters and the module. The definition is not given here, as WRIGHT expertise is still being gained.

```
System Example2
    Component SeqFile
        Port ReadNext [ with its protocol ]
        Port WriteNext [ with its protocol ]
        Spec [SeqFile specification]
    Component Filter
        Port StreamIn [ with its protocol ]
        Port StreamOut [ with its protocol ]
        Spec [Filter specification]
    Component Module
        Port ReadFile1 [ with its protocol ]
        Port ReadFile2 [ with its protocol ]
        Port RoutineDef [ with its protocol ]
        Spec [Module specification]
    Component Computation
        Port RoutineCall [ with its protocol ]
        Spec [Computation specification]
    Connector FileIo
        role Writer = [its role]
        role Reader = [its role]
        glue = [its glue]
    Connector ProcedureCall
        role Definer = [its role]
        role Caller = [its role]
        glue = [its glue]

Instances
    S : SeqFile
    F : Filter
    M : Module
    C : Computation
    SM : FileIo
    MC : ProcedureCall
    FM : FileIo
Attachments
    S.ReadNext as SM.Reader
    M.ReadFile1 as SM.Writer
    M.RoutineDef as MC.Definer
    C.RoutineCall as MC.Caller
    F.StreamOut as FM.Reader
    M.ReadFile2 as FM.Writer
end Example2
```

Figure 8. The second example in WRIGHT

5. 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 sinpler ones. As well as allowing for nested structuring, Darwin supports incremental structuring by extension. 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 (written in C++) would have to include calls to in and out methods defined for the **port** class, written in the Regis system in C++ [Magee 1994a], as shown in Figure 9. The C++ is 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, Steamlines is a class defined for the type of data being passed between the components. It serves the same purpose as the SIGNATURE("lines") statement in the UniCon version (Figure 5).

```
#include "CatFile.h"
#include "GlobalClasses.h"
#include <stdio.h>

CatFile::CatFile()
{
 StreamLines s;

 do {in.in(s);} while (!s.isAtEnd());

 FILE *f;
 f=fopen("paper.txt","rt");
 while(!feof(f))
 {
  char buffer[BUFFER_SIZE];
  fgets(buffer,sizeof(buffer),f);
  s.set(buffer);
  out.out(s);
 }
 fclose(f);
 s.markEnd();
 out.out(s);
}

#include "RemoveVowels.h"
#include <string.h>
#include <ctype.h>
```

```
int isVowel(char possibleVowel);
{
 possibleVowel=tolower(possibleVowel);
 return
   ((possibleVowel=='a')||(possibleVowel=='e')||
   (possibleVowel=='i')||(possibleVowel=='o')||
   (possibleVowel=='u'));
}

RemoveVowels::RemoveVowels()
{
 StreamLines s, outgoing;
 in.in(s);
 while(!s.isAtEnd())
 {
  char temp[BUFFER_SIZE]; temp[0]='\0';
  for (int ct=0;ct<strlen(s());ct++)
   if (!isVowel(s()[ct]))
   {
     temp[strlen(temp)+1]='\0';
     temp[strlen(temp)]=s()[ct];
   }
  outgoing.set(temp);
  out.out(outgoing);
  in.in(s);
 }
}
```

Figure 9. The actual C++ components used in the simple example

Now one could insert a pipe component between CatFile and RemoveVowels, and the Darwin program would be as in Figure 10.

```
component Environment
{
 inst
   s : System;
   stdIn: StandardInput;
   stdOut: StandardOutput;
   stdErr: StandardError;

 bind
   stdIn.yield -- f.in;
   f.out -- stdOut.receive;
   f.error -- stdErr.receive;
}

component Filter
{
 provide in<port StreamLines>;
 require out<port StreamLines>;
 require error<port StreamLines>;
}

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

component RemoveVowels: Filter;
component CatFile: Filter;
```

```
component System: Filter
{
 inst
   c: CatFile;
   r  RemoveVowels;
   p: Pipe;
 bind
   in -- c.in ;
   c.error -- error;
   c.out -- p.source;
   p.sink -- s.in;
   r.out -- out;
   r.error -- error;
}

component StandardError
{
 provide receive<port StreamLines>;
}

component StandardOutput
{
 provide receive<port StreamLines>;
}

component StandardInput
{
 require yield<port StreamLines>;
}
```

Figure 10. The simple example in Darwin

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 these 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 discrepency could not be detected. Finally, we note that the existance of a pipe connector in no way means that Darwin can make use of the pipe facility in the underlying operating system. This is not the case in UniCon, where the built in connectors can link up directly with whatever is available in the platform on which UniCon is running (in this case, Unix).

Pipe is a reasonably easy abstraction to insert into Darwin programs. However, the other connectors offered by UniCon have to be modelled laboriously. For example, shared data would require a locking mechanism implemented in C++, and the data could not be used naturally: it would have to be fetched via the message passing system first. So if X is a variable shared by the components A and B described in section 2, Unicon and WRIGHT could allow B to have statements such as

```
X := X + 1;
```

assuming that X is owned by A. All the protection associated with X would be taken care of by the shared data connector between A and B.

In a system connected with Darwin, we would need a component which has ports to lock, transfer and unlock the data. The code in B would look like this:

```
lock;
transfer.in (X);
X := X+1;
transfer.out (X);
unlock;
```

6. Related work

The original MILs relied on procedure call for interaction, or as Garlan calls it: Definiton/Use. Configuration languages conceived in the later eighties such as PCL [Dobbings 1993], Durra [Barbacci 1993] and Polylith [Callahan 1991] do not have connectors, but some do include a more general form of connection, such as the software bus [Purtilo 1994]. The original form of Darwin that was based on Pascal [Magee 1993] provided only one kind of port, but in principal the new version based on C++ classes should enable a greater variety of connections to be defined. In practice, as we have shown, these must be restricted to the underlying message-passing paradigm. More recently, 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]. In [Rice 1994] and [Magee 1994b] attempts have been made to provide formal models for configuration programmming languages, but neither of these places emphasis on connectors. The work that goes into the greatest depth formally so far is that of [Allen 1994b]. Mention should also be made of the recent extensions by the UniCon team [Shaw 1995a] 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**.

7. Conclusions

We set out to evaluate the value, properties and status of connectors in configuration programming, and to see whether they could be adequately modelled by a language which does not have them. The following was established:

1. Value. Connectors can be assesed in terms of their contribution to understandability and security of the resulting systems. Because they reveal the structure of the connection, they are a valuable architectural tool. Because the

connection can be checked in terms of its behaviour as well as the data it handles, they make a significant contribution towards security.

2. Status. A layered categorisation of languages with and without connectors was defined, and examples from each category dicussed.
3. Properties. A list of five properties was drawn up, and these are present in some form or other in the languages studied.

Finally, we present a comparison of the three languages with respect to certain factors which became evident as the research progressed (Figure 11). The first row indicates for noting that Darwin is heavily based on a C++ programming platform, and an accompanying environment called Regis [Magee 1994a]. UniCon runs presently in Unix environments and generates instructions for Odin, a unitlity similar to make. Wright has as yet to state where it will place itself.

	Darwin	UniCon	Wright
PL or OS base	C++ and Regis	Unix and Odin	any
Connector level	built-in paradigm	enumerated set	user-defined
Coupled	loosely to PL	close to OS and PL	none
Protocol given in	C++ classes	OS or PL mechanisms	Wright definitions

Figure 11. Comparison of Darwin, UniCon and Wright connectors

The connector level has already been discussed, and we can now see a consequence of it, as shown in the last two rows of the table. Since Darwin does not have connectors, it relies on the 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, as was illustrated in the example in Section 5.

UniCon on the other hand scores because it has carefully crafted a set of likely connected and has arranged for these to be directly linked into the existing constructs in the operating system or language. Just to show that the user-defined connectors are not always the best, we see that WRIGHT is almost the same as Darwin in that the protocols may be defined and visible, but it will be exceedingly difficult to tie these up with operating system constructs.

In conclusion, therefore, we can say that connectors are definitely a good idea. It is yet to be proven that they can be efficiently implemented and made use of, since both UniCon and Wright are still in the experimental stage. There are advantages and disadvantages to user-defined connectors, and it would seem that the best way forward would be to try to provide both user-definition and an enumerated set of common connectors.

8. Future work

The work described here is being continued as part of a larger project to investigate the developemnt of a new distributed computing environment for South African universities. The project is called Polelo, which is the Tswana word for communication, and involves a strong team of ATM experts. Our research will be to continue to define new connectors, to evaluate them in existing languages, and if possible to extend Darwin to accommodate a high-level form of connector.

Acknowledgements

The teams involved in Darwin, Unicon and WRIGHT have given willingly of their time and expertise in answering our questions for this paper. We thank in particular Greg Zelesnik, Jeff Kramer, Mary Shaw and David Garlan. Any errors in the work remain our own.

References

[Allen 1994a] Allen R and Garlan 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 Garlan D, *Formalizing architectural interconnection*, Proc. 16th Int'l Conference on Software Engineering (ICSE), Sorrento, Italy, May 1994.

[Barbacci 1993] Barbacci MR, Weinstock CB, Doubleday DL, Gardner MJ and Lichota RW, *Durra: a structure description language for developing distributed applications*, Software engineering journal, **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] J R Callahan and J M Purtilo, *A packaging system for heterogeneous execution environments*, IEEE Trans. Soft. Eng. 17 (6) pp 626– 635 June 1991

[Dean 1995] Dean T and Cordy JR, *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 1993] Magee J, Dulay N and Kramer J, *Structuring parallel and distributed programs*, Software Engineering Journal **8** (2) 73–82 March 1993.

[Magee 1994a] 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.

[Magee 1994b] Magee J, Eisenbach S and Kramer J, *System structuring: a convergence of theory and practice?,* internal report, Department of Computing, Imperial College, London, 1994.

[Purtilo 1994] J Purtilo, *The Polylith software bus*, *ACM Trans. on Prog. Lang. and Sys.*, **16** (1) 151–174 January 1994

[Rice 1994] Rice MD and Seidman SB, *A formal model for module interconnection languages*, IEEE Trans. Soft. Eng. **20** (1) 88–101, 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 1995a] Shaw M, DeLine R and Zelesnik G, *Abstractions and Implementations for architectural connections*, unpublished manuscript, March 1995.

[Shaw 1995b] 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.