

Characteristics of modern system implementation languages

J M Bishop and R Faria
Computer Science Department, University of Pretoria
Pretoria 0002, South Africa
E-mail: jbishop@cs.up.ac.za

Abstract: Systems are written in systems implementation languages. What characterizes such languages in the mid-1990's? This paper identifies the typical environment that a system is being targeted for these days – as opposed to twenty years ago – and emphasizes how this radically affects the language in which development proceeds. It then considers two directions that systems development is taking. The first relies heavily on design tools and methodologies, which are the means for describing the underlying structure of the system. The second interposes an additional layer of description between the design steps and the programming language, and maintains the structure here. The rest of the paper focuses on these intermediate languages, identifying their characteristics and advantages and looking at two examples, Darwin and UniCon. In conclusion, we highlight the (sometimes surprising) results that have emerged so far, and predict the way in which modern systems implementation languages could move.

1 Introduction

Systems implementation languages, like programming languages in general, have evolved through various generations. In the early days, building a system involved starting at the beginning and programming until one reached the end. Certainly in the sixties and seventies, systems software was of manageable size to make this possible. Taking the example of a compiler as an archetypal system, in the later 1960s, a BASIC compiler and file manager could be written in 6000 lines of assembler. In the 1970s, the original recursive descent Pascal compiler which was 2000 of Pascal grew to 4800 when it was rewritten with an analyzer/generator interface. Interestingly enough, the assembler program used units and separate compilation, whereas the Pascal programs did not, this not being a supported feature at the time. Pascal was perhaps backward in this respect, since C certainly had *include* and *make* facilities to ease both the programmer's task and the time spent on system development. As another example, the Burroughs B5700 operating system written in Burroughs Extended Algol was no more than an inch thick. It also did not use libraries.

Figure 1 summarizes these simple days. The salient point is that the structure of the system was primarily embodied in the actual SIL code. If one was lucky, and the language was Pascal or an Algol variant, and written according to the structured programming practices which were newly in vogue, then it was still possible for a single person to find their way around a system, maintain and modify it. But for many programmers, a system meant sheaves of assembler, commented or not, documented or not, and working or not.

Improving this situation was one of the main activities of computer scientists from the mid-1960s. The bid to make SILs more effective then followed several routes. **Structured programming** was the first movement, and it revolutionized coding practice in the 1970s. New SILs had to make sure they had the right programming constructs to support it. Thereafter, **stepwise refinement** and type checking were promoted, both via the strong voice of the Algol 68 supporters and through the success of Pascal [Wirth 1977].

By the 1980s, **type checking** in languages was becoming universally accepted, even for systems programmers who might still grumble about its cost, and the new feature was **modularity**. Languages such as Ada, Modula-2 and C++ could provide at their own level the linkage facilities which were previously available only at assembler level. Moreover, type checking was now possible across modules, which it was not in C or in the versions of Pascal that eventually supported separate compilation.

A second major concern in the 1980s was reliability of compilation. Compilers at the time were mostly produced by computer companies and were notoriously error-prone. The computer science community responded to the challenge to produce reliable compilers with a two pronged attack. The first

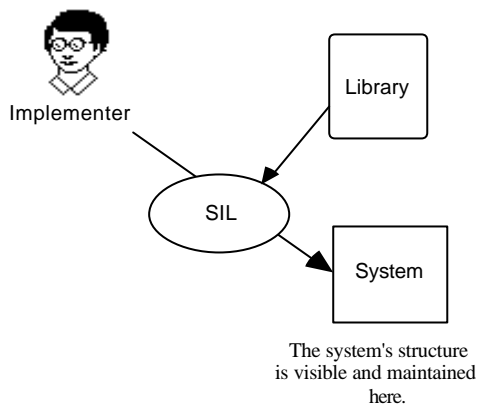


Figure 1. *Early use of SILs*

concentrated on compilation techniques and especially the development of **compiler tools**, and the second aimed to **stream-line** the language so that there was simply not as much chance for error. But with hardware becoming more sophisticated, the demands for more language features are hard to gainsay. This was particularly true in the real-time, parallel and distributed environments, where synchronization, memory management and communication had to be accommodated. The resulting languages were often larger than their sequential counterparts.

In the 1990s, languages will be put to use in the construction of reliable, efficient systems that must last for 20 years or more. With the cost of software still mushrooming, system managers now demand nothing less. With the compiler problem mostly solved, and stream-lining already at its limits, it is clear that further improvement in modern system implementation languages requires a fresh approach. What is this approach to be?

We answer this question in the next five sections. Section 2 looks at the changing environment in which SILs will be used. Section 3 identifies two distinct approaches to the development of large and complex systems. Section 4 picks up the second of these, which defines a new type of SIL called a system structure language (SSL) and gives it a definitive role in the software development cycle. Examples of the new genre are given and a preliminary assessment is attempted. Section 5 compares the two languages and Section 6 concludes by listing the benefits of the new SSLs and the challenges that still need to be faced.

2 The changing environment

Factor 1. Distributed computing is here to stay

Distributed and real-time programming is generally regarded as a “systems” activity. Thus languages developed especially for these areas are generally able to maximize the performance of features, at the expense of user-friendliness. However, the third community which uses multi-computers – the parallel programmers – understandably look askance at the contortions required to express themselves in occam and even C++. Hence the enduring nature of Fortran and its libraries, and the blossoming of new communication tools such as the Parallel Virtual Machine (PVM).

The challenge of distributed computing is therefore not just expressibility, but ease of use.

Factor 2. Computing power is and is not a scarce resource.

One could almost call this Paradox 2. Considering storage first, it is an axiom that there is never enough. Whether working on a file server or on a stand-alone machine, there comes a time when the disk fills up, or “special” memory overflows. The difference between the old days and now is that this is happening so much faster. It is quite usual for a new version of a system to approach twice the size of the old one. Falling back on the old excuse that “memory is cheap” is no longer sufficient when the total cost of upgrading *all* machines in a firm is considered. System managers are not going to accept that they must throw more hardware at their system every year. Size must be brought under control once again.

As far as speed goes, computers may well have cycles to burn, but human beings do not, and there is now a definite move towards demanding instantaneous results. If not instantaneous, then at least we would want the time taken to deal with a problem to be proportional to the extent of the solution.

Factor 3 Development is operating system intertwined

The requirement that a really good SIL provide a development environment for portable software has diminished of recent years. The front runners that emerged from the 1980s achieved their success by standing on the shoulders of good operating systems. The prime example is C which runs most efficaciously on top of Unix, but there are also Oberon on Oberon [Wirth and Gutknecht 1992] Orca on Amoeba [Tanenbaum *et al* 1993] and occam on the transputer development system (TDS) [INMOS 1990]. The notable exception is Ada, but perhaps many of Ada’s woes can be attributed to its need to produce portable systems, and not to rely on the benefits provided by a particular operating system.

Not being portable does not imply not being multi-platform: indeed it is mandatory these days that a system be able to migrate among the big name hardware platforms. But it is now more acceptable to specify either that the environment must also be present (for example Unix in one of its many guises) or the particular version for that platform be loaded (as Microsoft does with its mainline packages).

Factor 4. Tailored languages are increasingly attractive

The days of general purpose languages being the correct medium for every system are now past. Several excellent special purpose SILs have been developed for example TXL (which focuses on text translation) [Cordy and Carmichael 1993], Eli (for compiler construction) [Gray *et al* 1992] and Darwin (for system composition) [Magee *et al* 1994]. Like any tailor-built software, these languages offer efficiency, leverage and a short learning curve if chosen for an appropriate problem. By their existence, though, they make the systems programmer’s life more difficult because the criteria for choosing between them and a general purpose offering are subjective. Factors such as availability, life-span, cost, user base and support, may well favor C++ or Ada, but as we shall see, a level-by-level approach to SILs is one of the ways in

which we can control software development, and special-purpose languages provide such a level.

3 Approaches to modernizing SILs

Consider then, that the SILs we need in the next decade will have to produce distributed systems and will be operating fairly closely to a well-known operating system and environment. While they will have more memory and cycles than ever before to play with, they have to be able to control the size and cost of systems and their development. This concern for cost control exists at all phases of development through to execution and possible later reconfiguration. We now identify two main approaches to evolving such SILs.

3.1 Design methodologies and tools

In the first approach, we recognize that there have been tremendous development and investment in design tools in the past decade [Booch 1994, Rumbaugh *et al* 1991]. The primary purpose of these tools is to enable the system design team to capture the ideas and requirements of the user more easily and accurately, and to embody these naturally in a structure which can serve as the basis for the implementation of the system. In the now classic system design methodology, the SIL comes in at the end, and its impact on the design is kept at bay. This is not to say that such a delayed decision is a bad thing: with a choice of SILs, as explained in the previous section, it is wise to have more information at hand before committing to a particular language.

What the reliance on design tools does produce is an overall system where the structure is embodied in the design itself. It is here that the objects, methods and relationships are established and written down. It is at this layer that they are available to be read, studied and pinpointed for change. (At least, this is how the methodology is meant to work: in practice, tweaks to the code often happen.)

The intended advantage of having the structure kept by the design tool is that it should be at a high enough level to be a means of communication between the user, designer, and implementers (Figure 2). In practice, the latter two are usually at ease with the notation, but the user understands very little unless it is interpreted. From the point of view of the SIL, therefore, this is not a determining factor. Figure 2 also shows that the existence and use of a library of reusable components is an integral part of modern development using SILs.

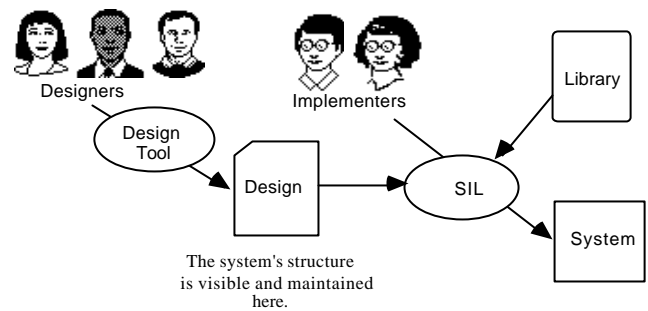


Figure 2. *The design tool approach to enhancing SILs.*

The disadvantage of this approach, from the point of view of SIL development once again, is that improvements and benefits of a new SIL can be negated if there is no nice fit between it and the design output. For an example, a system designed using structured analysis can certainly be implemented with a powerful object-oriented SIL such as C++, but the full range of class features may well not be called upon. One then gets a feedback loop where the choice of SIL influences the choice of design tool and the two become closely coupled. Separation of concerns and ease of change are both jeopardized.

Nevertheless, this approach is currently the most widely used and is enjoying the greatest investment from software engineers both in industry and academia. It must be noted, however, that the major developments tend to focus on the development of new and better tools, rather than on SILs.

3.2 The system structure language approach

Given that the influence of design tools on SILs and vice versa can be detrimental to the evolution of a system, we can consider a second approach which is emerging from several quarters. We continue to make maximum use of design tools and of the best available SIL, but in between we insert a new layer: that of a system structure language (SSL). The design proceeds as before, but now we have a choice. Instead of the design driving the implementation, dictating which modules have to be written and how they are connected, we consider instead the scenario of Figure 3.

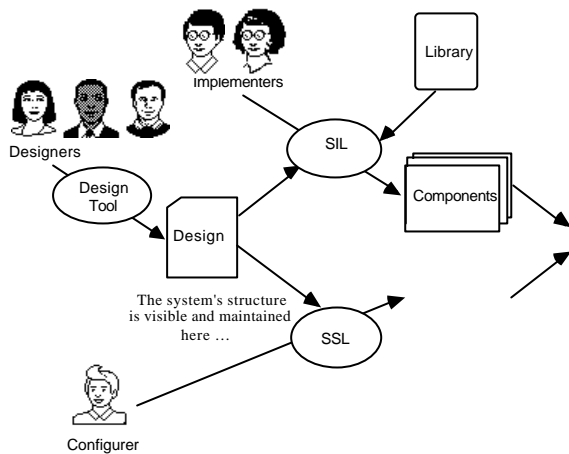


Figure 3. The system structure language approach to enhancing SILs

The implementation can proceed in both a decompositional and a constructive way. In the first case, the design drives the implementation of components, but in the second case the design also drives the creation of a separate structure description of the overall system, and from this new components and connections may emerge. These latter components and connections would have more to do with the finer detail of the implementation, and especially with its existence in a distributed environment. The advantages of the additional layer of development are several. Firstly, the visibility of the system is enhanced. The output of the SSL – called a configuration here – is a concise encapsulation of the components and connections of the system – but at the system level, rather than at the design tool level. Thus it can take into account issues related to communication and processor allocation.

Secondly, the ability to reconfigure the system without changing either the design or the implemented components is made possible. All that is required is further SSL programming, and this should always be shorter and more concise than changes to either of the other modes.

Thirdly, we are beginning to realize that the composition of systems from reusable components is an activity that does not always have to be undertaken from scratch. Therefore it is reasonable to imagine that a library of software configurations can be built up, and that these can form a reusable base for future use. Examples that spring to mind are the specification of dynamically created tree structures, of pipelines, of voting algorithms for the replacement of faulty processors, and of the host of classic concurrent policies for handling mutual exclusion and synchronization. Because these are now well known, their structure can be embodied in SSL code, so that the system builders simply provide the components [Callahan and Purtilo 1991], [Magee *et al* 1994], [Shaw and Garlan 1993].

As always, the arguments against adding an extra layer in the software development process must center on the efficiency of the resulting product. Initial results have indicated that the configuration layer can effectively be wiped away once the final

system is up and running, thereby incurring minimum cost [Magee *et al* 1994, Shaw *et al* 1993].

Having identified two of the major movements for bolstering systems programming, we now concentrate on the second. What do these SSLs look like? What are their responsibilities and characteristics? How are they to be used in practice, and how are we to evaluate them? And most importantly, what examples are there in use at present?

4. System structure languages

4.1 Terminology and introduction

First of all, we must mention that like SIL, SSL is not a term with universal currency. In fact, I have used it here simply to emphasize the *structure* aspect of its role in the software process. It has been used before [Sommerville 1992] but probably the better known term which encompasses a similar group of languages is MIL or module interconnection language [DeRemer and Thomas 1976, Prieto-Diaz and Neighbours 1986]. In the distributed computing community, we also have configuration language [Kramer 1990] and system description language [Barbacci *et al* 1993]. Some use the term architectural description language [Shaw *et al* 1993], and a recent workshop was devoted to interface description languages [Wing 1994]. Rather than try to stipulate a common term, we stick to SSL and hope the reader will do the necessary term translation.

4.2 Responsibilities

An SSL has the **responsibility** to enable the following activities:

1. The **composition of components** written in a system programming language into higher level components, and ultimately a system. This property of hierarchical composition is vital: a single level of constructive “glue” will not suffice.
2. The provision for a **variety of communication mechanisms** between components. This property can be provided either by means of SSL level “connectors” or by means of access to mechanisms at the SIL level.
3. The description of **frameworks of structure** that embody and identify common abstract software patterns, and their instantiation for given sets of components.
4. The ability to specify **alternative configurations** as well as the circumstances that will trigger them.
5. The **mapping of components onto distributed processors** in as hardware-independent a manner as possible.

4.3 Characteristics

An SSL has the following **characteristics**:

1. It is a separate and **different** language to a SIL, usually declarative in nature.
2. It upholds the **typing** of the languages with which it associates.
3. It is potentially able to deal with components from a variety of SILs, as well as “dusty deck” components, i.e. it is **heterogeneous**.
4. It may have a **graphical** equivalent notation.
5. It is usually implemented on top of a specific operating system and **environment**.
6. There is **no overhead** in efficiency compared to a system composed by hand.

4.4 UniCon

We consider first UniCon, proposed by a group under Mary Shaw at CMU [Shaw *et al* 1993]. UniCon’s purpose is to be an existence proof for a more general model of architectural description language. UniCon stands for Universal Connector, and a strong focus of the work is on providing access to different connection mechanisms (responsibility 2). Figure 5 gives an example of part of a UniCon program taken from the work of Shaw *et al*. It implements the KWIC indexer depicted in Figure 4.

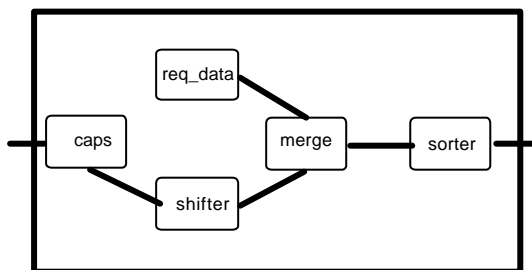


Figure 4. KWIC indexer system

The INTERFACE section describes the outer shell of the KWIC indexer system. The PLAYERS at this level are the input, output and error streams. In total, KWIC operates as a filter, so it is identified as such an architectural pattern from the start. To implement the indexer, we need five other components, and these are instantiated from their type definitions, held elsewhere. Notice that the line:

```
USES caps INTERFACE upcase
```

uses the keyword INTERFACE rather than COMPONENT when making the instantiation. In the same way, the three connectors used in the system, P, Q and R are instantiated with the keyword PROTOCOL, which introduces the connector’s interface.

Next we establish the connections. UniCon uses BIND for the connections that are external to the component itself, and CONNECT for gluing together internal components. Finally, as with the connectors P, Q and R for the internal components

(which were all Unix pipes), we establish that KWIC itself forms a Unix pipe, with its two external points.

Overall, UniCon embodies all of the characteristics listed in the previous section, and the first three responsibilities. It does not, as yet, provide for hardware mapping or for reconfiguration. It is implemented on top of the Mach operating system, and makes use of the Mach Interface Generator. For real-time systems, each component is an RT Mach thread, and the whole system is a heavy weight process [Kitayama *et al* 1993].

One added feature in UniCon is support for analysis tools and specification notations developed by others. It was deemed very important by the designers that the configurations that they carefully programmed in UniCon could be independently tested by some other means. Specifically, UniCon supports rate monotonic analysis for detecting scheduling problems in meeting real-time deadlines.

4.5 Darwin

Darwin was developed at Imperial College London. Darwin, following on from its ancestor Conic, belongs to the configuration

```

COMPONENT KWIC
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
/* First instantiate the parts to use */
USES caps INTERFACE upcase
USES shifter INTERFACE cshift
USES req-data INTERFACE const-data
USES merge INTERFACE converge
USES sorter INTERFACE sort
USES P PROTOCOL Unix-pipe
USES Q PROTOCOL Unix-pipe
USES R PROTOCOL Unix-pipe

/* Next associate players of some parts to players of the
interface. */
  BIND input TO caps.input
  BIND output TO sorter.output

/* Finally describe the way KWIC is built from parts. This
can be done by directly associating players with roles. */
  CONNECT caps.output TO P.source
  CONNECT shifter.input TO P.sink
  CONNECT shifter.output TO Q.source

```

```

CONNECT req-data.read TO R.source
CONNECT merge.in1 TO R.sink
CONNECT merge.in2 TO Q.sink

/* Complete the connections */
  ESTABLISH Unix-pape WITH
    merge.output AS source
    sorter.input AS sink
  END Unix.pape
END IMPLEMENTATION
END KWIC

```

Figure 5. UniCon program for a KWIC indexer system

language arm of SSLs. As such, its focus is very much on the successful mapping of systems onto distributed hardware [Magee *et al* 1994].

Darwin is modest in its approach to variety. The latest versions of Darwin actively support responsibility 2, i.e. the provision of a variety of communication mechanisms, but do so through links to C++ template classes. As such, the language is virtually tied to C++, and does not meet characteristic 3

```

#include "upcase.dw"
#include "cshift.dw"
#include "constData.dw"
#include "converge.dw"
#include "sort.dw"

component kwic
{
  inst
  caps: upcase;
  shifter: cshift;
  reqData: constData;
  merge: converge;
  sorter: sort;

  bind caps.output -- shifter.input;
  bind shifter.output -- merge.in1;
  bind reqData.output -- merge.in2;
  bind merge.output -- sorter.input;
}

```

Figure 6 Darwin version of the KWIC indexer

(heterogeneity). Earlier versions of Darwin used additional primitives to provide communication mechanisms, but this meant that the choice was restricted by Darwin itself. Now, the choice is restricted only by the ingenuity of the C++ programmer. Figure 6 shows the same KWIC indexer program as before rephrased by the authors in Darwin. The Darwin source is noticeable shorter than the UniCon version. The reason for this is two-fold.

Firstly, UniCon has the separate interface section which identifies the external players in the component. In Darwin, these are just ports and portrefs as normal.

Secondly, UniCon instantiates three separate connector components to link shifter, req_data and merge, and Darwin uses direct point-to-point connections. Looking further, we see that this brevity on the part of Darwin is carried through to the lower level components. Figure 7 shows the cshift component in both languages.

The VARIANT statement in the UniCon source of Figure 7(a) is important. It identifies where the actual code for this component lies. In Darwin, this linking is done by include statements output by Darwin itself when it creates header files (see below). The strength of an SSL is that identifying a sub-component can be altered without changing the component itself or any others that it will be bound to.

Consider a further Darwin example, that of a readers-writers package, written by the authors. Here the upper level component, rw, is given in Figure 8. We can replace the scheduler being used by a different one simply by changing two lines here – the inclusion of the Darwin code for it, and its instantiation as *s* via the inst command in line 9.

The Darwin for the scheduler component is shown Figure 9. Part of the assistance that Darwin gives to the implementer is the creation of appropriate header files corresponding to each component. These can then be used as a guide to writing the program itself. Typically, such headers are about 60 lines of C++, and Figure 9 shows the beginning of one. The headers enable Darwin to link up to its environment, Red, which provides the underlying support for execution of components as light weight threads within a single Unix process [Crane and Twidle 1994].

```

COMPONENT cshift
INTERFACE IS
  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
  VARIANT cshift IN "cshift"
  IMPLTYPE (Executable)
  END cshift
END IMPLEMENTATION
END cshift

```

(a) Unicon

```

component cshift
{

```

```

    provide input (port stream);
    require output (port stream);
}

```

(b) Darwin

Figure 7. UniCon and Darwin descriptions for a low level component

```

#include "reader.dw"
#include "writer.dw"
#include "scheduler.dw"

component rw
{
    const int readerMax = 10;
    int writerMax = 5;
    array r[readerMax]: reader;
    array w[writerMax] : writer;
    inst s.scheduler(readerMax, writerMax);
    forall i:0..readerMax-1
    {
        inst r[i];
        bind r[i].startRead -- s.startRead;
        bind r[i].stopRead -- s.stopRead;
    }
    forall i:0..writerMax-1
    {
        inst w[i];
        bind w[i].startWrite -- s.startWrite;
        bind w[i].stopWrite -- s.stopWrite;
    }
}

```

Figure 8. A Readers-Writers configuration in Darwin

```

component scheduler( int maxReaders, int maxWriters)
{
    provide startRead <port notifier>;
    provide stopRead <port notifier>;
    provide startWrite <port notifier>;
    provide stopWrite <port notifier>;
}

#include <dw.h>
class scheduler : public process {
public:
    port<notifier> startRead;
    port<notifier> stopRead;
    port<notifier> startWrite;
    port<notifier> stopWrite;
    scheduler(int maxReaders, int maxWriters);
};
... plus Darwin generated code for interfacing with Regis

```

Figure 9 A low level Darwin component from file scheduler.dw and its C++ header output

An additional change from earlier versions of Darwin is that there is now no support for reconfiguration (responsibility 4). This is a decision made by the designers based on scepticism of the current state of the art. At a later stage, the language could be extended.

4.5 Frameworks

Responsibility 4 lists reusable frameworks as essential for an SSL. The UniCon work concentrates on examples which describe common connection patterns between components, such as Unix-pipes, filters and remote procedure calls. As more work is done, more complex patterns can be built up. In Darwin, the emphasis is on abstract structures which can be instantiated for actual components. A good example of such structures is a generic binary tree component [Magee *et al* 1994]. A bnode component type is defined as

```

component bnode {
    provide upper <<alpha>>,
           lower <<alpha>>;
    require result <<alpha>>;
}

```

It provides two input ports (as they are called) and requires an output port. The actual message types to be transmitted on the ports will be provided later, and is specified here as <<alpha>>. The bintree component is parameterized for the number of components and their type (itself a component). Its header is:

```

component bintree (int n, component bnode)

```

Instantiating such a generic component for a simple integer adder component such as:

```

component add {
    provide A<port, int>, B<port, int>
    require sum <port, int>
}

```

can be done by

```

inst paradd : bintree (N, add);

```

Darwin is a versatile SSL which could well be used outside of its parent community of distributed systems.

5. Assessing languages

System structure languages provide an additional layer of software which reveals structure, reuses existing architectural patterns and enables swift and checked reconfiguration. As such they are a significant addition to the systems scene. In assessing their possible future impact, it is useful to look back at the criteria used in the past to assess languages. In [Shaw *et al* 1981], the programming-in-the-large features for languages for software engineering were assessed according to the following list:

- decomposition of the system
- definition and use of abstractions

- enforcement of independence
- assembly of systems from components
- physical independence of modules
- checking module linkage
- understandability.

This list ties in very well with the philosophy of SSLs. In particular, the last criterion, which the authors identify as “the most outstanding characteristic needed in a maintainable system” is a prime reason for the existence of SSLs in the first place. [Feuer and Gehani 1984] add to this list

- ability to access routines written in other languages
- provision for concurrent programming
- existence of reliable and efficient compilers

which once again brings us closer to an SSL. In particular, the heterogeneity issue is raised, as well as concurrency. Turning to concurrent languages especially, we note that [Stotts 1982] includes the following list of macro criteria, in addition to those specifically related to process handling:

- ease of verification
- real-time support
- ease of use.

All of these are embraced by UniCon in particular. Finally, we note the list of that master language designer, Niklaus Wirth, compiled back in 1977 [Wirth 1977], which adds the element of cost.

- a complete definition without reference to compiler or computer
- a modularization facility
- conciseness and clarity of description
- a totally reliable compiler
- compiling at reasonable speed
- generating efficient code
- reasonably predictable execution cost
- simple and effective interface to the environment.

These comparisons were generally aimed at assisting users in choosing a language. The work presented here certainly has that as an overtone. As such, it is helpful to summarize the features of the two languages according to the list of responsibilities and characteristics identified here (Figure 10).

What we still need to do is investigate other quantitative criteria such as efficiency as well as qualitative criteria such as conciseness and ease of use. A more complete study has been done for configuration languages which will form the basis for comparing SSLs in general [Bishop 1994]. Shaw and Garlan [1994] have also set out some desiderata for languages at this level.

		UniCon	Darwin
Responsibilities	composition	hierarchical	hierarchical
	communication	connector components	C++ template classes
	frameworks	yes for connectors	yes
	reconfiguration	not yet	no
	mapping to hardware	not shown	yes
Characteristics	different to SILs	yes – Ada like, declarative	yes – C++ like, mostly declarative
	strict typing	yes	yes
	heterogeneous	yes	no
	graphical notation	yes	yes
	OS and environment	yes – Mach RT	yes – Unix
	overhead in efficiency	minimal	minimal

Figure 10. *Comparison of UniCon and Darwin*

6. Conclusions

Developing a number of small programs has given us a modest understanding of the Regis environment (release 0.4.4, March 1994). We experimented with a textual based version of Darwin, running on the PC-based Linux operating system.

Darwin descriptions clearly and explicitly describe a system’s architecture. This feature facilitates the documentation of the system architecture. Configuration information is kept separately in special Darwin descriptions, allowing the system to be easily reconfigured at a later stage. After finalizing the system’s architecture, the Darwin compiler generates C++ header files containing the required interface for each component in the

system. The Darwin compiler also creates the main program used to initialize and drive the whole system. A side-effect of this automated process in Regis is that the system dictates the structure of the primitive C++ components. Old programs will have to be re-written in order to work in the Regis environment. Fortunately after converting an older C++ system, we have ascertained that the rewriting process may not be too tedious.

UniCon was built as an existence test and as such can not be considered to be in its final form. However, it has extended the boundaries of SSLs considerably, with its connector components, its insistence on heterogeneity and its provision for analytical tools. However, the syntax is clumsy: too much has to be specified too often. From working with small examples, we would recommend that this problem can be largely overcome by defining sensible defaults. Although we have not yet had an opportunity to use UniCon, we would hope to be able to apply this idea and to continue to develop trial systems in parallel with Darwin, to gain more comparative evidence.

SSLs are being developed at a steady rate. They provide an opportunity for systems programming to become more visual, more structured and more amenable to change. While more research is needed to find the ideal SSL, it is certain that candidates exist and that greater usage of these languages can be expected in the future. The research being undertaken by the authors seeks to evaluate languages continuously, defining criteria and providing test beds and reports of findings. As they are made known, such results should assist systems programmers in choosing the right tool for the job.

References

- Barbacci M R and several others, *Durra: a structure description language for developing distributed applications*, Software Engineering Journal **8** (2) 83–94 March 1993.
- Bishop Judy M, *Languages for configuration programming: a comparison*, University of Pretoria Technical Report 94/04, submitted for publication 1994.
- Booch G, **Object oriented design with applications** 2nd ed, Benjamin Cummings 1994.
- Callahan J R and Purtilo J M, *A packaging system for heterogeneous execution environments*, IEEE Trans. on Software Engineering **17** (6) 626–635 June 1991.
- Cordy J R and Carmichael I H, *The TXL7 programming language syntax and informal semantics*, Software Technology Laboratory Report, Queen's University, Kingston, Ontario, available on ftp from TXL@qucis.queensu.ca.
- Crane S and Twidle K, *Constructing distributed Unix utilities in Regis*, Proceedings of the 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, 183–189, available from the IEEE, 1994.
- DeRemer F and Kron H, *Programming-in-the-large versus programming-in-the-small*, IEEE Software Engineering **2** (2) 80–86 1976.
- Feuer A and Gehani N, *A methodology for comparing programming languages*, in their book **Comparing and assessing programming languages**, Prentice-Hall, 197–208, 1984.
- Gray R W, Heuring V P, Levi S P, Sloane A M and Waite W M, *Eli: a complete, flexible, compiler construction system*, CACM **35** (2) 1221–131, February 1992.
- INMOS, **Transputer Development System**, Prentice-Hall, 1990.
- Kitayama T, Mercer C W, Nakajima T, Savage S, Tkuda H and Zelenka J, *Real-time Mach 3.0 User Reference Manual*, School of Computer Science, CMU, Pittsburgh, August 1993.
- Kramer J, *Configuration programming – a framework for the development of distributable systems*, Proceedings of the IEEE International Conference on Computer Systems and Software Engineering (CompEuro 90), Israel, May 1990
- Magee J, Dulay N and Kramer J, *A constructive development environment for parallel and distributed programs* Proceedings of the 2nd International Workshop on Configurable Distributed Systems, Pittsburgh, 4–14, available from the IEEE, 1994.
- Prieto-Diaz R and Neighbours J M, *Module interconnection languages*, Journal of systems and software **6** (4) 307–334, November 1986.
- Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorensen W, **Object-oriented modeling and design**, Prentice-Hall 1991.
- Shaw M, Almes G T, Newcomer J M, Reid B K and Wulf W A, *A comparison of programming languages for software engineering*, Software Practice and Experience **11** (1) 1–52 1981.
- Shaw M and Garlan D, *Characteristics of higher-level languages for software architecture*, unpublished manuscript, Carnegie Mellon University, 1993.
- 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*, unpublished manuscript, Carnegie Mellon University 1993.
- Sommerville I and Thomson R, *Configuration specification using a system structure language* in Proceedings of the 1st International Workshop on Configurable Distributed Systems, London 1992.
- Stotts P D, *A comparative survey of concurrent programming languages*, SIGPLAN Notices **17** (10) 76–87 October 1982.
- Tanenbaum A, Kaashoek F M F and Bal H E, *Parallel programming using shared objects and broadcasting*, IEEE Computer **25** (8) 10–20, August 1992.
- Wing J M, ed. Proceedings of the *Workshop on Interface Definition Languages*, Portland January 1994, in SIGPLAN Notices **29** (8), August 1994.
- Wirth N, *Programming languages: what to demand and how to assess them*, **Software Engineering** ed. Ron Perrott, Academic Press, New York, 155–173, 1977, also in **Comparing and assessing the programming languages Ada, C and Pascal**, eds A Feuer and N Gehani, Prentice-Hall, 245–261, 1984.
- Wirth N and Gutknecht J, **Project Oberon: the design of an operating system and compiler**, Addison-Wesley 1992.