Imperial College of Science, Technology and Medicine
University of London
Department of Computing

A Configuration Language for Distributed Programming

Naranker Dulay

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering of the University of London, and for the Diploma of Imperial College of Science, Technology and Medicine

February 1990

Abstract

Most conventional languages for distributed programming combine the notions of algorithmic programming, non-local interaction and program structuring within a single programming language. This leads to programs that hide their structure and that lack the ability to support dynamic program modification at a high level of abstraction.

This thesis presents an alternative approach to developing distributed programs that separates the structural organisation of a distributed program from the algorithmic implementation of its parts and leads to distributed programs that are easier to describe, construct and manage. This is achieved by using a separate language, termed a configuration language, to describe the structure of a distributed program and for its construction and reconfiguration.

The configuration language is declarative and supports both modular construction and type safe interactions. The language allows hierarchic, replicated, variant, parameterised, switching and recursive program structures to be specified. An interactive version of the language includes further declarations to dynamically modify the structure of running programs. The implementation model for the language supports incremental development. It allows program sub-structures to be compiled in isolation on possibly heterogeneous hosts and then safely configured into the running program.

This thesis presents

- the configuration language for the description of the structural topology of a distributed system programmed as a set of configuration-independent processes.
- an abstraction mechanism, called a group module for structuring systems as a hierarchy of interacting subsystems.
- a method for transforming the hierarchic structure (and possibly recursive) of a program into a more efficient heterarchic structure.
- techniques for efficiently compiling distributed programs constructed with this language into distributable units called nodes.
- techniques for distributing and reconfiguring the nodes of a distributed program across a heterogeneous collection of processors.

A set of example programs that demonstrate both the utility and expressive power of the language are given. The language has been implemented within the context of the Conic environment and is in use at several institutions around the world.

Acknowledgements

This is the 'thank you' page. As such, it is often skipped by the reader, but to the author it is perhaps the most important page of all. Without the help and friendship of the people mentioned below, the work in this thesis would not have taken on its final form.

First and foremost, I would like to thank my thesis supervisor Jeff Kramer, for his guidance, encouragement and patience throughout the development of this work.

The comments of Morris Sloman and Jeff Magee who took time to read a draft of this thesis are also much appreciated.

The work in this thesis is the result of many fruitful collaborations and discussions with members of the Distributed Programming Group at Imperial College. I am especially indebted to Jeff Kramer, Morris Sloman and Jeff Magee for giving me the opportunity to work in such a stimulating environment and for supporting me in my work. I would also like to especially thank Kevin Twidle, Keng Ng and Anthony Finkelstein for their encouragement and support. Special thanks also go to Anne O'Neill for her prompt and efficient service. In addition collective thanks go to the remaining members of the group, past and present.

The system described in this thesis is the result of the cooperative efforts of a number of people. In particular Jeff Magee was responsible for the design and implementation of the Conic run-time executive, the virtual target concept, the dl program, the spy utility, and the implementation of Batcher's bitonic sort described in this thesis. Kevin Twidle was responsible for the design and implementation of the interactive configuration manager (Iman) and the node server. Keng Ng was responsible for the design and implementation of ConicDraw, the graphical configuration manager.

This work and my stay at Imperial College has been supported by funding from British Coal, and the Science and Engineering Research Council.

To Satpal and Jes

Contents

Figures

Chapter One Introduction

1.1. INTRODUCTION

The structure of distributed programs and the extent to which that structure is visible significantly affects the clarity of distributed programs [Ossher87]. Clarity, in turn, can favourably affect other important goals for distributed programs such as modularity, reusability, reliability and modifiability as well as aiding distributed program design and debugging [DeRemer76, Horning73]. For these reasons, distributed program structuring and the clarity of its specification are of central importance to distributed program development.

Distributed programs consist of collections of smaller components. The precise nature of these components depends on the nature of the distributed programming system being used, but typically comprise executable components, such as processes, modules, and procedures, as well as non-executable components such as data variables, files and communication buffers . The organisation and interaction of the components of a distributed program together constitute the structure of a distributed program and as the demands for distributed programs increases, so does the requirement for mechanisms and notations specifically for organising and managing the structural complexity of resultant programs [Randell86].

Most current languages have some mechanisms for structuring programs. For example, many languages include import-export declarations which describe and control the interactions between modules and have data structuring mechanisms which specify the organisation and interaction of the data used within a program. Such mechanisms are quite good for small sequential programs that consist of an amorphous collection of modules, but are bad at describing the richer variety of program structures possible for distributed programs. New, more flexible program structuring mechanisms are needed for describing the structural complexity of distributed programs.

As DeRemer and Kron point out in their seminal work [DeRemer76], the process of structuring programs, which they term "programming-in-the-large", is an intellectually different activity from algorithmic programming. Many current languages for distributed programming tend to combine the notions of program structuring, algorithmic programming, and non-local interaction within a single programming language. Unfortunately, this leads to programs that hide their structure and are harder to design, construct, and debug as well as being more difficult to manage and dynamically modify.

A number of distributed programming systems have recognised the different processes involved by embodying notations for program structuring within a separate linguistic framework. Such languages have been called a variety of names in the literature including module interconnection languages [DeRemer76], graphical interconnection languages [Weide82], network specification languages [LeBlanc82] and configuration languages [Dulay87, Lee86]. This thesis focuses on the provision of an integrated program structuring language and toolkit for the development of distributed programs.

1.2. THE PROBLEM ENVIRONMENT

Compared to single processor computer systems, distributed computer systems promise many important advantages, for example high availability, expandability, performance, resource sharing, de-centralisation, and cost-effectiveness [LeLann81]. Higher availability results from the replication of resources such as processors and file systems that is a common feature of distributed systems. The failure of a single resource in the system need only result in graceful degradation. Similarly expansion of a running distributed system can carried out with little or no effect on existing users. Distributed systems can in general be extended to a much greater degree before the onset of performance bottlenecks. Claims for infinite expandability have been made for some distributed architectures e.g. hypercubes [Sietz85]. Many distributed systems are also designed to offer higher performance, e.g. speed, throughput, problem-solving capacity. This is achieved by parallel execution of programs. The above characteristics make distributed systems the preferred approach to the future computing requirements of many organisations.

Many flavours of distributed system are now available, for example, shared memory multi-processors such as the Sequent Balance [Thakkar88], tightly coupled point-to-point computers such as the NCUBE [Hayes86], and loosely coupled networked computers such as clusters of Sun workstations. In the research community, on-going work on fine-grained machines such as dataflow machines, graph reduction machines and connectionist machines may lead to highly parallel systems. Modern machines also feature greater connectivity ability, and the growing trend towards opens systems is likely to accelerate as demands from users for sharing and exploitation of computing resources increase. Computer systems can no longer operate in "splendid" isolation and must be programmed for coordinated operation [Sloman87].

A uniform approach to programming these heterogeneous distributed systems will clearly be needed. One way of tackling this, is by reconsidering the requirements for distributed programming and seeing if a simpler, more natural programming paradigm can be developed.

1.3. THE CONIC TOOLKIT

The work in this thesis has been implemented and is available as part of the Conic toolkit at Imperial College [Dulay87]. The Conic toolkit has been developed over the last 10 years and is in use in institutions around the world. The toolkit provides a comprehensive set of language and run-time tools for program compilation, configuration, execution and debugging in a distributed environment. Conic programs may be run on a set of interconnected host computers running the Unix operating system and/or on target machines with no resident operating system.

The starting point for the work in this thesis was the original Conic system completed in 1984 (called Conic84 in this thesis) and described in [Kramer83, Magee84]. This provided a rudimentary configuration language for structuring process control applications running on a set of bare targets. The system also lacked adequate language and tool support for the development of large distributed programs running in a heterogeneous host-target environment. The work in this thesis helps remedy these problems by providing more scalable solutions for distributed program development.

1.4. THESIS OBJECTIVES

The objective of the thesis is to show that the single language style of distributed programming is ill-suited to large-scale distributed programming, and that a dual language approach that separates distributed programs into structural components and programming components is both clearer and efficiently realisable. The thesis presents a integrated language and set of tools and techniques to demonstrate this. In particular the thesis presents:

- a new advanced declarative configuration language for the description of the structural topology of a distributed system programmed as a set of configuration-independent processes.
- a new abstraction mechanism, called a group module for structuring systems as a hierarchy of interacting subsystems. The mechanism combines modularity with expressive power, while the structures produced by the mechanism are capable of being dynamically reconfigured.
- techniques for efficiently and safely compiling distributable units called nodes from large numbers of separately compilable sub-components. The techniques employ a new symbol file design for efficiently representing the interfaces of separately compilable sub-components. Symbol files are also used to track object files in the host file system.
- a new technique for performing type extension checks in constant time rather than linear time.
- an algorithm for transforming the hierarchic (and possibly recursive) structures of a group module into a more efficient heterarchic structure at run-time.
- techniques for distributing and reconfiguring the nodes of a distributed program across a heterogeneous collection of processors.
- tools to support the construction and debugging of distributed programs.

1.5. THESIS OUTLINE

This organisation of this thesis is as follows: Chapter 2 reviews the general properties of distributed programs and discusses the approaches taken to programming them. It then proceeds to identify the key requirements for producing flexibly-structured distributed programs. Existing approaches are critically examined in relation to these requirements. Chapter 3 describes an advanced fully implemented language, termed a configuration language for producing flexibly-structured distributed programs. The language is based on a new structuring mechanism, termed a group module that satisfies the requirements identified in Chapter 2. Chapter 4 demonstrates the power and applicability of the language with a variety of examples. Chapter 5 presents an implementation model for realising the configuration language on a heterogeneous distributed platform. Chapter 6 describes how large numbers of separately compilable units are efficiently and safely compiled into runnable units called nodes. Chapter 7 describes the run-time techniques for efficiently elaborating program structures at run time, as well as techniques for distributing program structures onto processors. Chapter 8 presents conclusions, a critical evaluation of the configuration language and suggests directions for future work. A definition of the configuration language is given in Appendix I. A definition of the syntax of the configuration language compiler's symbol files is given in Appendix II.

Chapter Two Distributed Programs

2.1. INTRODUCTION

A distributed program can be considered as a set of distinct nodes which are spatially separated, and which interact with one another by exchanging messages. Distribution results in concurrent execution, non-determinism, and non-neglible message transmission delays between nodes compared to the time between events occurring within a single node.

Distributed programs distinguish themselves from other categories of programs by being subject to de-centralised decision making, to continuous change and evolution, to the need for node negotiation and co-operation, and to the sudden unavailability of resources [Hewitt85].

Techniques for improving reliability and availability, for locating services or objects given some symbolic or indirect name, for migrating data and program code are primarily associated with distributed programs. Distributed programmers often incorporate techniques that are designed to perform well in the presence of high communication latency, being optimised for this property at the expense of other costs that would be relatively more significant in a system having lower communication latencies. Distributed programmers also tend to optimise for low communications bandwidth.

Distributed applications are diverse, they range from computation-intensive applications such as, parallel algorithms for solving the optimisation problems, to data-oriented applications, such as the provision of database services, transaction-processing, and mail services, to control-oriented applications, such as the co-ordination of robots in a factory plant.

Although the underlying reasons for many distributed programs are better performance, fault-tolerance, perhaps the most important reason for their use, is that they often offer a simpler or more natural solution to the problem.

Constructing, maintaining and understanding distributed programs, like any large program, is difficult. Understanding its structure is an important first step, but this is itself a complex task. There is a clear need for well chosen program structuring mechanisms to provide assistance in this task.

This chapter identifies the key requirements for producing flexibly-structured distributed programs. These requirements are also examined in relation to other work. The chapter first overviews the models and program structuring mechanisms used in distributed programming. The actual language developed to meet these requirements is described in the next chapter.

2.2. DISTRIBUTED PROGRAMMING MODELS

Distributed programming systems can be considered to belong to one of two models, termed the implicit model and the explicit model.

The implicit model is characterised by the automatic compilation of a program into distributable parts. Examples include vectorising and parallelising compilers of sequential languages such as Fortran [Kuck77, Albert88] and C [Quinn88] and parallel implementations of functional languages such as Hope [Moor82] and Lisp [Larus88]. In addition programming systems in this model normally require special-purpose hardware for efficient execution, for example, the Connection Machine [Hillis85] and Flagship [Watson87]. Although functional programming languages are capable of yielding maximal concurrency, they lack the ability to handle non-determinism and cannot effectively model history-sensitive computations required in distributed programming. Logic programming appears to be even more poorly suited to programming distributed systems [Hewitt85, Kahn88].

The explicit model is characterised by the explicit programming of a distributed program of interacting parts. Examples include message-passing systems such Argus [Liskov84], CSP [Hoare84], SR [Andrews88], Actors [Agha86], and Parlog86 [Ringwood86], shared memory systems such as Linda [Carriero89], and hybrid systems such as Unity [Chandy88] that can be transformed to use message-passing or shared memory. In addition programming systems in this model are normally extended with extra mechanisms for concurrency and remote interaction such as processes, remote procedure calls, message-passing primitives, mailboxes, tuple-spaces, logic variables and difference lists.

2.3. PROGRAM STRUCTURING MECHANISMS

Program structuring mechanisms can be classified according to the "grain" of atomic parts that they manipulate. For imperative programming languages, procedures could be considered a fine-grain program structuring mechanism since they group together a set of atomic statements. Modules then act as medium-grain program structuring mechanism for procedures and interact by export-import declarations which provide the names of directly callable procedures. Most imperative programming languages do not, however, provide any further mechanisms for structuring modules into larger granules.

Object-oriented programming languages provide a more hierarchical mechanism, the class, which can be used to structure both methods (the fine-grain mechanism) and classes themselves [Stefik86]. Classes interact by calling the inherited methods of their parent classes.

Functional programming languages provide a different mechanism altogether, the high-order function. This can be used to compose very powerful and general functions from simpler ones [Backus78, Hughes88]. The requirement for modular program structuring mechanisms still exists however [MacQueen86].

Most distributed programming systems belonging to the explicit model provide concurrent processes and some mechanism for organising them, although many of these mechanisms are just extended forms of modules, classes or abstract data types, for example Argus provides guardians [Liskov83] and SR provides resources [Andrews88].

Some distributed programming systems have aimed to increase program structuring ability by providing more general structuring mechanisms based on explicit components interconnection. Examples include agents in CCS [Milner80], stations an networks in Conic84 [Magee84], subsystems in Dicon [Lee86], nodes in DPL-82 [Ericson82], translations in DTL [Hughes83], tasks in Durra [Barbacci88], agents in Garp [Kaplan88], shells in HPC [LeBlanc85], teams and systems in Lady [Wybrabietz85], nets in Muppet [Muhlenbein88], networks in Netsla [LeBlanc82], nodes and configurations in RNet [Coulas88], and boxes in Stile [Weide82].. Earlier examples include C/Mesa [Mitchell79], Intercol [Tichy79], MIL75 [DeRemer76], and PCL [Lesser79].

The need for powerful structuring mechanisms has also been addressed in related areas. For example (i) the hardware description languages Fable [Ossher83] HISDL [Lim82], and Strict [Campbell86] provide abstraction grids, structures, and blocks respectively (ii) the design method Mascot 3 [Bate86] provides subsystems, (iii) the specification system Statemate [Harel88] provides statecharts.

The mechanisms proposed however, fail to adequately meet one or more key requirements for structuring distributed programs identified in the next section.

2.4. REQUIREMENTS

The key requirements for structuring distributed programs are summarised below:

Separate Structural Specifications
The specification of a distributed programs structure must be separated from its programming and not included within it.

Conceptual Simplicity and Clarity
Structural specifications must be clear and readable, and act as an aid to readers in understanding the program.

Declarative specification
Structural specifications must be declarative, they should describe program structures, and not detail how those structures are to be used.

Executable specification
The structural mechanism must be realisable, that is, amenable to implementation and tool support. Specified program structures must be built automatically without further transformations by the user.

Expressive power
The structuring mechanism must be sufficiently expressive to be able to specify a rich set of program structures.

Modularity
The structuring mechanism must promote modularity through re-usability and information hiding.

Concurrency
The structuring mechanism must be able to express concurrent program structures.

Scalability
The structuring mechanism must be able to handle large programs. Structure specifications must not grow unmanageably large and complex as program size

increases.

Interaction Safety
The structuring mechanism must ensure that unintended interactions are not introduced into the program.

Adaptability
The structuring mechanism must be capable of supporting modifications to the structure of a running program.

Independence
The structuring mechanism must be capable of supporting different programming languages.

Although the requirements for structuring distributed programs summarised above are motivated in part by the nature of the Conic environment available at Imperial, they are not peculiar to the Conic environment or to distributed programming. Large, complex programs arise in many contexts and a structuring mechanism that could be used to specify their structure also, would be of general use.

We now discuss each of the above requirements in more detail.

2.4.1. Separate Structural Specifications

The structure of a large distributed program can be exceedingly complex, yet an appreciation of it is crucial to an understanding of the program. It is, in fact, so important, that one generally tries to understand the overall structure before examining the details. The many advantages of "structured-programming" derive from the fact that good structure is an important aid to people in their understanding and managing of complexity. These advantages only apply if the structure is readily apparent.

The fundamental advantage of having explicit and separate structural specifications is that they provide a means to communicate program structure between people [DeRemer76]. The designer of a distributed program will generally have a mental picture of the programs structure. That picture affects his understanding of the program, his design decisions, and ultimately the details of the program itself. A reader of a distributed program, particularly a reader unfamiliar with it, is likely to derive great benefit from knowing the structural model that the program designer used.

By separating out the structural specification of a distributed program other advantages accrue, for example, (1) the structural specifications of a program can be perused separately from the algorithms, and used as an index into the program, since structural information is grouped together, rather being scattered throughout the program (2) structural specifications can be wholly or partially constructed before the program is written, and then used to direct program development, (3) the structural specification is suited to providing a convenient repository for information about a program, such as documentation, (5) the structural specification is a valuable representation of a program, of use to software tools that manipulate programs, for example, compilers, browsers and command shells, debuggers, and simulators (e.g digital circuit simulators), visualisation tools, and load balancing programs. Structural specifications can also act as input for expert systems.

Separation of distributed programs into a structural part and a programming part can lead to programs that are easier to design, construct and debug, programs that easier to distribute and manage, as well as programs that are easier to change.

The separation of the structure of a program from its programming has been adopted and applied in several systems, for example: Dicon, DTL, Durra, Garp, Lady, C/Mesa, Muppet, Netsla, and RNet. The separation of structural concerns can also be found in Fable, HISDL, Mascot 3, Statecharts and Strict.

The formal system CCS [Milner80] incorporates a structuring mechanism, but this is embedded within the behaviour of CCS agents, and not separated from it. DPL-82 and Stile employ hierarchic program structuring based on dataflow graphs, but each allows programming statements to be embedded: Dijkstra's guarded command language [Dijkstra75] in Stile, and Pascal in DPL-82. HPC requires special user-written processes to define and manage the structure of programs. No notation is provided for structural descriptions.

The functional languages DL [MacQueen86] and Peeble [Burstall84] address structural issues by extending the type system of the language to include module types. This provides a formal semantic framework for analysing the underlying notions of program structuring mechanisms, but does not clearly separate them from the programming.

Modular languages such as Ada [Ada83] and Modula2 [Wirth82]; distributed languages such as Argus, Linda, NIL [Strom83] and SR; declarative languages such as Prolog [Clocksin81], Scheme[Abelson85] and Miranda [Turner86]; object-oriented languages such as C++ [Stroustrup86], Emerald [Black87], and Smalltalk [Goldberg83] all fail to provide explicit and separate structural specifications for programs.

The formalism in Durra also includes non-structuring specifications based on path expressions and real-time logic for timing and ordering of message queues, and behavioural specifications based on Larch [Guttag85]. Ideally the structuring mechanism should be orthogonal to other kinds of specification, and capable of independent consideration and development. The inclusion of such formalisms leads to configuration languages that are harder to use well, harder to define semantically and harder to implement.

2.4.2. Conceptual Simplicity and Clarity

Hoare has written of programming languages [Hoare68] that: "The most valuable feature of a programming language is that it provides the programmer with a conceptual framework which enables him to think more clearly about his problems and about effective methods for their solution; and it gives him a notational technique which enables him to express his thoughts clearly". This remark is equally true of structuring mechanisms and their realisation in distributed programs. Structure specifications should be clear and readable, and act as an aid to readers in understanding programs. Ideally they should be sufficiently simple to write and manipulate so as to enable non-programmers to construct their own programs in a do-it-yourself manner with little effort.

Conceptual simplicity is best achieved in systems that economise on concept(s) and/or that have a sound underlying mathematical framework. A good example which combined both of these principles is CCS.

Many systems fail to provide simple structuring mechanisms, for example C/Mesa makes heavy use of defaults, and although these can greatly simplify program structure, they can also obscure a great deal of the structure concerned. C/Mesa also suffers from the multiplicity of ways of specifying the same program structure.

2.4.3. Declarative Specification

The configuration language must be declarative, it should enable the specification of a distributed program's structure without requiring information on how the structure is to be realised. Declarative specifications are more concise than non-declarative ones and more amenable to analysis, transformation, augmentation, and manipulation. Most structuring mechanisms are declarative for these reasons. The notions of execution state or control should not be encompassed by the language. Different strategies should be applicable to elaborating the structure, for example, lazy evaluation.

2.4.4. Executable Specification

The division of a distributed program into structural parts and programmed parts should constitute a complete program that is capable of efficient execution on existing distributed computers.

Non-executable structural specifications act as design specifications only, and require an additional transformation before a complete program can be constructed. Furthermore, this transformation needs to be carried out each time it is desired to change the structure of a program. Having an executable mechanism alleviates the need for further transformations and enables rapid prototyping of distributed programs.

The languages Dicon, Durra, Lady, Muppet, Netsla and Rnet have been implemented on distributed computer systems, while Garp, C/Mesa, and Stile have been implemented on single processor systems. DTL has been implemented using an abstract virtual machine interpreter, and subset of CCS called LL [Thorelli85] has been used as an object module link-loading language.

2.4.5. Expressive Power

Program structuring mechanisms must be sufficiently rich to cope with the wide diversity of distributed programs that arise in practice. The structuring mechanism of configuration languages should aim for expressive power that increases the applicability or flexibility of the approach.

Examples of useful program structures include: hierarchic structures, replicated structures (arrays), parameterised structures, variant structures, recursive structures (e.g trees), switching structures, and dynamic structures. Of these, hierarchical structuring is the most essential, without it large programs cannot be well-structured [Dijkstra71]. The absence of any of the other forms of structuring will limit the applicability of the structuring mechanism.

Most existing configuration languages have hierarchical structuring but tend to lack the remaining forms. Durra for example has hierarchical structuring and a form of dynamic structuring, but lacks replicated and recursive structures. Rnet only allows two levels of hierarchic structuring, while C/Mesa lacks hierarchic structures altogether. The structuring mechanism of DTL provides replication and recursion plus three composition operations, pipeline composition, disjoint parallel composition, and cyclic composition which it is claimed are sufficient for structuring any concurrent program. Since CCS is a mathematical theory, it allows the full power of mathematics to be used if desired, e.g subscripting, conditional expressions, operator definitions, recursion.

2.4.6. Modularity

Configuration languages must promote the proven notions of re-usability and information hiding [Parnas72b]. This can be achieved by making program structuring mechanisms modular and abstract. Modularity encourages top-down design and testing of modular parts in isolation. With modularity, large distributed programs can be specified and studied in increasing order of complexity while reducing an explosion of details, leading to better comprehensibility.

The program structuring mechanism should also support abstraction and ensure that interactions take place through well-defined interfaces that lead to context-independent (loosely coupled) modules. Interfaces should be minimal and define only needed information and nothing more, further no information should be provided about the underlying structure behind the interface [Parnas72a].

The absence of modularity and abstraction leads to large monolithic programs that are hard to understand, hard to maintain, and non-reusable.

Most configuration languages provide a controlled interface to their structuring mechanism thus supporting context independent modules.

2.4.7. Concurrency

The division of programs into a collection modules was an important step in increasing programmer productivity, but the style of sequential processing carried out by a such programs is clearly inadequate for the kinds of distributed computer systems that are available today, that range from fine-grained dataflow computers, to large-scale computing networks. Program structuring mechanisms should therefore not limit the degree of concurrency which a distributed program may require. Sequential programming solutions are often just special cases of more general concurrent solutions, in which programs can be written more easily and be better understood as set of simple concurrently interacting components instead of as one sequential component.

Systems which require sophisticated compilers and-or specialised hardware suffer performance loss by not being able to easily extract or express the right level of concurrency for efficient execution.

2.4.8. Scalability

Scalability is an important criteria for evaluating program structuring mechanisms. Aesthetically, scalable mechanisms are more elegant and robust. Practically, scalable mechanisms imply less work in the future adapting to new technologies. Ideally structure specifications should not grow unmanageably large and complex as structure size and complexity increases. Good expressive capabilities are obviously needed, but also support for complexity management techniques, such as divide-and-conqueor, and separation of logical concerns. Techniques that are successful in dealing with small programs often break down in the face of the complexity of large programs, so scalable solutions are important.

2.4.9. Interaction Safety

The utility of a configuration language increases enormously if provision is made in the structuring mechanism to prevent unintended and undesirable interactions between the parts of a distributed program [Horning73]. Allowing precise controls on the interactions can contribute significantly to reductions in the complexity of distributed programs.

A popular technique is to provide type checking of the interfaces of program parts when structuring programs. This is provided for example in HPC, C/Mesa, Lady, Netsla, and Stile. Assertions and constraints also help ensure against undesirable interactions. Rnet provides for consistency checks of timing specifications although these are too primitive to be of practical use.

2.4.10. Adaptability

All programming systems support change by recompilation and re-execution. Some distributed applications also require support for dynamically modifying the structure of programs at run-time. The structuring mechanism must be capable of supporting such modifications.

Both Netsla and Durra allow pre-planned changes to be specified within a configuration specification activated by the satisfaction of an event. CCS does not support change, although lazy elaboration of recursive agents can be used to simulate a dynamic structures. HPC has an elaborate model for dynamically re-structuring of programs, but this relies on special user-supplied processes to control changes. In Garp components can replace themselves with a new set of components when they terminate. Systems that allow program structures to be manipulated as values in programs such as NIL possess dynamic re-configuration ability, but fail in the requirement for separate structural specifications, which should extend to structural modifications.

The ability to perform unplanned (or evolutionary) modifications to the structure of a running program is also important [Kramer85, Kahn88]. Most programming systems in existence lack such ability.

2.4.11. Independence

The configuration language should be independent of particular programming languages. This increases its generality and utility as language for building distributed programs, and could allow programs to be composed of components written in different programming languages.

Most structural approaches are unfortunately coupled to a particular programming language, for example, DTL is coupled to a programming language based on attributed translation grammars, Garp is coupled with LISP, Lady is coupled with CSSA, RNet is coupled with Concurrent Euclid.

The structuring mechanism of Durra is programming language independent, although message transformation functions need to explicitly included in the structural specification. Dicon is notable for allowing program parts to be implemented in C, Lisp or Prolog.

Distributed program structures should also be specified independently of the actual architecture on which they will be run on. Mapping the parts of a distributed program to computing resources should be deferred until program execution is needed. Dicon and Rnet fail in this regard, as the mapping of programs is made a function of their program structuring mechanism.

2.5. CRITICAL SUMMARIES OF OTHER SYSTEMS

2.5.1. CCS

CCS provides a formal framework and notation for reasoning about about concurrent programs. CCS is declarative but fails to provide a separate configuration language for structuring programs. Processes in CCS are called agents. Three program structuring operators are provided for agents: a composition operator for linking agents, a restriction operator for hiding agent interfaces, and a relabelling operator for renaming agent interfaces. CCS programs can be constructed hierarchically, modular interfaces are not provided. Rather the language requires that agents hide interfaces by the extensive use of the restriction operator. Since CCS is a mathematical theory, it also admits mathematical devices such as universal quantifiers, subscripting, conditional expressions, and operator definitions. It would be of interest to know of any distributed implementations of CCS and of what features were dropped or included in order to get a practical system. Although dynamically evolving configurations can sometimes be described by recursive definitions, CCS fails to adequately address the issue of dynamic structuring. The major contribution of CCS is in fact, its rich underlying theory of concurrent systems.

2.5.2. Conic84

Conic84 provides a rudimentary configuration language for structuring distributed process control applications running on bare targets. Three different levels of structuring are provided: modules which declare a set of concurrent tasks, stations which interconnect modules and can be distributed, and networks which interconnect stations into a distributed program. Conic84 fails to support arbitrary levels of hierarchical structuring. No support is provided for parameterised, replicated, variant, switching, or recursive program structures. Conic84 also fails to support more than one programming language. The major contribution of Conic84 is its support for the online reconfiguration of program structures.

2.5.3. Dicon

Dicon provides a configuration language for structuring distributed real-time programs. At its lowest level are distributable granules, which can be written in C, Lisp or Prolog. Granules can be interconnected to form subsystems, or interconnected with subsystems to form a system. It is not clear whether subsystems can nest. No support is provided for parameterised, replicated, variant, switching, or recursive structures, or for modifying the structure of running programs. The language is also complex since it allows non-structural specifications such as resource requirements, real-time constraints, process assignment constraints and process control statements to be included.

2.5.4. DPL-82

DPL-82 employs hierarchic program structuring based on dataflow graphs, but fails to provide a separate configuration language. DPL-82 programs are structured into nodes. Nodes are written in a mixture of Pascal and Lisp. Pascal is used to write the computations performed by a node, while Lisp is used for creating, connecting, activating and terminating nodes. Lisp is also used to send results from one node to another. No support appears to be provided for variant, switching or recursive structures, or for modifying the structure of running programs. DPL-82 also lacks a distributed implementation.

2.5.5. DTL

DTL is a novel language for concurrent programming. It includes both structural components called concurrent translations, and programming components called sequential translations. Concurrent translations are used to specify networks of sub-translations (either concurrent or sequential). Parameterisation, replication and recursion plus three composition operators: pipeline, parallel, and cycle are provided for expressing network descriptions. These it is claimed are sufficient for structuring any concurrent program. DTL does not appear to support switching structures or variant structures however. DTL also fails to support unplanned modifications to the structure of running programs. The interface of a sequential translation consists of a typed input stream and a typed output stream. Sequential translations are written as a set of production rules over the input and output streams of the translation. DTL has been implemented using an abstract virtual machine interpreter but lacks a distributed implementation. The choice of an abstract syntax-directed programming language also seems very limiting.

2.5.6. Durra

Durra provides a configuration language for distributed real-time applications such as robot control. Durra supports hierarchical structuring of interconnected networks of processes called tasks. Parameterised, replicated, variant, switching, and recursive program structures are not supported however. Durra configurations can specify functional and timing specifications but Durra lacks tools to check, analyze or enforce such specifications. These specifications also increase the complexity of the language. Durra allows pre-planned changes to be specified within a configuration specification activated by the satisfaction of some event. Again these appear not to be implemented. Durra is claimed to be programming language independent, although only C is supported. Durra programs can run on heterogeneous machines, although message transformation functions need to be explicitly supplied and configured between processes running on heterogeneous machines. This implies prior knowledge of the physical configuration on which programs will run and leads to machine-dependent configuration specifications.

2.5.7. Garp

Garp provides a formalism (part graphical) for writing concurrent programs with reconfigurable structures. Garp employs graph grammar specifications for describing the legal structures a system can evolve to. At the lowest level, processes called agents can terminate themselves by performing a rewrite action, that replaces the agent with a new structure described by a graph grammar production. Existing links to and from the terminating agent are relinked to agents in the replacement structure. The interface of the replaced agent must therefore match the interface of the replacement structure. The top level of a Garp program is normally a dummy agent that immediately terminates and replaces itself by the actual structure of the program as specified by the top-level graph grammar production. Garp only supports agents in Lisp and lacks a distributed implementation. Garp also lacks modularity. It is not clear whether Garp supports replicated, variant or switching structures. Garp is interesting for its treatment of preplanned changes to its program structure, although pure interconnection changes are very cumbersome and inefficient. They require a new agent to unnecessarily replace the existing one, plus the coping of the state variables from the old agent to the new via parameters. Unplanned reconfigurations to the structure of a program are not handled in Garp.

2.5.8. HPC

HPC does not provide an explicit configuration language, but rather a model for dynamically reconfiguring hierarchic process structures. HPC programs are structured hierarchically into shells. Each shell can include one privileged process called a controller that is responsible for all high-level operations on components in the shell. Controllers configure components, perform interaction checks, create sub-controllers, interact with parent controllers and child sub-controllers, and ensure application consistency. Controllers must be written by the programmer. If a shell does not include a controller, its components are transparent to the parent shell and its controller. If a shell does include a controller, the components of the shell are hidden from the parent shell. Although the use of user-written controller processes to manage levels of the hierarchy may lead to greater flexibility, it also increase the burden of programming needed to build an application, as well as introduce further sources of error.

2.5.9. Lady

Lady is a configuration language for distributed operating systems. Three different levels of structuring are provided: modules which declare a set of processes and monitors, teams which interconnect modules, and systems which interconnect teams into a distributed program. In addition to direct interconnections, Lady also supports indirect interconnections via logical buses. Lady fails to support arbitrary levels of hierarchical structuring or replicated, variant, switching, and recursive program structures. The structure of running programs can be manipulated but only from within the programming language. Lady thus fails to preserve the separation of programming from configuration. Unplanned reconfigurations to the structure of a program are not supported.

2.5.10. Muppet

Muppet provides a configuration language for parallel programming. Muppet programs are structured hierarchically into process nets and specified graphically. Replicated structures can be defined in predefined regular topologies such as grids and trees. It is not clear whether programmers can specify their own topologies, and if so whether this is done declaratively. Muppet also allows weights to specified for processes and interconnections. These are used as hints when mapping processes to processors. Muppet supports two programming languages, Concurrent Modula2 and Occam. Occam program running on transputers, must however, perform their own routing. No support is provided for modifying the structure of running programs.

2.5.11. Netsla

Netsla is configuration language for distributed programs. Netsla programs are structured hierarchically into networks of processes. Parameterised, replicated, variant, switching, and recursive program structures are not supported however. Netsla is notable for handling dynamic modifications to the structure of a program entirely at the configuration level. This is done by the inclusion of a complex sublanguage for reconfiguration based on event handling. Reconfigurations taking place at different levels require serialisation however. Netsla also fails to support unplanned modifications or more than one programming language.

2.5.12. Rnet

Rnet provides a rudimentary configuration language for distributed real-time programs. Two levels of structuring are provided: nodes which declare a set of processes and networks which interconnect the processes in one node to the processes in other nodes. Rnet fails to support arbitrary levels of hierarchical structuring. No support is provided for parameterised, replicated, variant, switching, or recursive program structures, or for modifying the structure of running programs Rnet also fails to support more than one programming language and lacks modularity.

2.5.13. Stile

Stile employs hierarchic program structuring based on dataflow graphs, but fails to separate this from the programming. No support appears to be provided for parameterised, replicated, variant, switching or recursive structures or for modifying the structure of running programs. Stile also fails to support more than one programming language, and lacks a distributed implementation.

2.6. CHAPTER SUMMARY

This chapter has argued for a structural approach to distributed programming. The approach abstracts out the structure of a program into a form that can be used to design and construct the program as well as be used to reconfigure the program. The separation of a distributed program into two different levels of abstraction, one for program structure and one for program implementation provides a conceptual framework in which distributed applications can be clearly specified and easily developed.

The properties and requirements of a structuring mechanism to capture structural specifications were identified and discussed. First and foremost that the mechanism be conceptually simple, and include the minimal number of concepts required to enable distributed programs to be clearly structured. The mechanism must also be expressive enough to handle a wide variety of distributed applications. The mechanism should also be capable of scaling up for large applications. The mechanism should promote modularity while preserving safety when structuring. The mechanism should be capable of efficient realisable. In addition structural specifications should be declarative, and independent of particular programming languages and architectures.

This chapter has also examined existing approaches to structuring and building distributed programs, and shown how these approaches fail to fulfil the requirements identified.

Chapter Three The Configuration Language

This chapter describes a fully realised configuration language for structuring distributed programs. This language aims to meet the requirements for structuring distributed programs identified in Chapter 2. The concepts and mechanisms in the language are described and small examples of their use given. Diagrams are used to show the structures and reinforce the descriptions. The description of the language is followed by a summary of the language and on how it successfully meets the requirements set out in Chapter 2. A definition of the language is given in Appendix I.

3.1. PROCESSES

Processes (tasks) are used in our configuration language as the smallest components in structuring distributed programs. The choice of such autonomous components is based on their suitability for distributed programming [Kramer85]. Other kinds of component may be more appropriate in other contexts. Parallelism within processes is left for programming language compilers to identify and translate for given target machines.

Process components are defined in suitable process programming languages, but minimally must make available to the configuration language a process name. For example, the process specification:

```
Task Module P
        Internals
End
```

defines a process[1] component with name P:

1 Historically the configuration language has always used the word **task** instead of process.

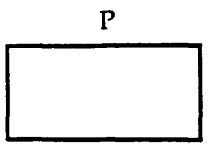


Fig 3.1 Process Definition

Component definitions act as templates or types, from which one or more component instances can be created. The keyword **module** is used to emphasise the modular nature of process definitions.

Given a set of named process definitions, P1, P2, and P3:

```
Task Module P1          Task Module P2          Task Module P3
        Internals               Internals               Internals
End                     End                     End
```

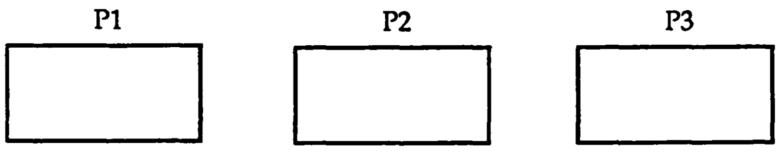


Fig 3.2 Multiple Process Definitions

we can declare a set of process instances X1, X2, X3, with the configuration specification:

```
Use P1; P2; P3;
Create X1 : P1;  X2 : P2;  X3 : P3;
```

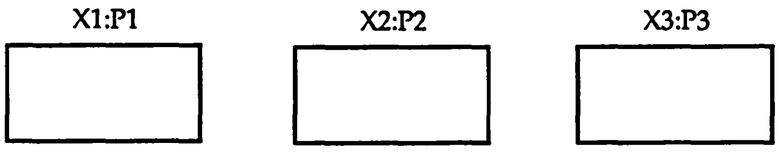


Fig 3.3 Instantiation of Processes

The identification of component definitions is termed **context definition**. Context definition serves to make available to the configuration, a set of component definitions, in this case, process definitions, which can used for declaring sub-component instances within the configuration.

The declaration of component instances is termed **component instantiation**. The ability to create more than one instance of a component is a highly desirable requirement and so the component types used in our configuration languages can all be *multiply instanstiated,* as in:

```
Use P1;
Create X1 : P1;  Y1 : P1;  Z1 : P1;
```

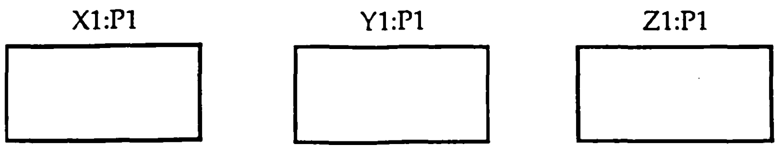


Fig 3.4 Multiple Instantiation of a Process

Configuration specifications of this form are conceptually simple, clear, and declarative. They do not however, explicitly expose the underlying interaction structure of a distributed program. The need to show such structure in configuration specifications is important as it allows processes to defined in a *configuration-independent* way. Configuration-independent processes can be written without knowing which processes they will interact with.

3.1.1. Direct Process Binding

The *interaction structure* of a distributed program can be specified by binding processes together. However, approaches where the binding of processes is embedded within the component programming language are less amenable to reconfiguration. The alternative to direct process binding is indirect process binding.

3.1.2. Indirect Process Binding

Two approaches to indirect process binding have been explored, the *process network* approach and the global or *shared memory* approach. The first is the natural analogue of actual computer networks, and is adopted in our configuration language. The process network approach has also be modelled in functional programs [Turner87] and concurrent logic programs [Shapiro84].

The shared memory approach as exemplified by the Linda system [Carriero89], totally uncouples processes. The basis of Linda is a logically-shared associative memory through which distributed processes communicate. Processes never exchange information directly but rather write to, and read from a global associative memory. Because of its conceptual simplicity and elegance, Linda is an appealing approach, particularly for applications where the processes which send data do not care which processes are to receive it or at what time. Ensuring that unintended interactions do not occur, is left to the programmer.

The implementation of logically-shared associative memory, in a distributed environment can be highly inefficient compared to the process network approach where the communication structure of processes is known. Doubts also exist as to whether global memory systems can be made to work efficiently and reliably for distributed systems with large numbers of processors. Since Linda can easily be simulated in the process network model by providing one or more processes to act as the logically-shared memory, and binding all other processes to these processes it appears that the network model is more general.

By making interaction structure and interaction safety a configuration concern, and by making process binding indirect, leads to distributed programs as reconfigurable networks of loosely-coupled, configuration-independent processes.

3.2. BINDINGS

The interaction structure of distributed programs can be specified by enumerating the possible process interactions. For example

```
Link  X1 to X2;
      X2 to X3;
      X2 to X4;
```

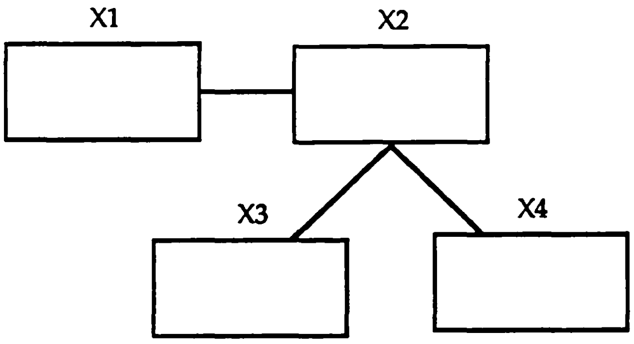


Fig 3.5 Binding of Processes

This is a simple, yet powerful approach to structuring distributed programs. Configuration specifications of this form do not however meet the requirement for interaction safety. In order to restrict and prevent unintended interactions, the declaration of the services provided by and/or required by individual components must be made, and checks done to ensure that process requirements and process provisions are bound safely. Having such precise controls contributes to reductions in software errors.

3.3. PORTS

Ports are used to explicitly identify the *interaction points* of a component. Ports are declared within component definitions, and implicitly instanstiated during component instantiation. Component can have has many ports as required.

Given a process P1 with port Q1, and component P2 with port Q2:

```
Task Module P1              Task Module P2
        Port Q1                     Port Q2
End                         End
```

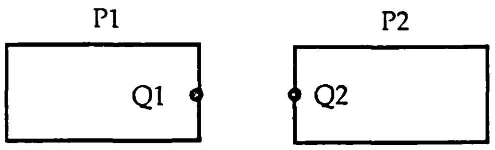


Fig 3.6 Process Ports

we can declare and bind instances X1 and X2 of these process definitions as follows:

```
Use P1; P2;
Create X1 : P1; X2 : P2;
Link X1.Q1 to X2.Q2;
```

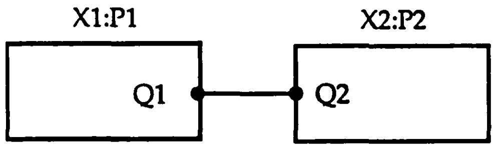


Fig 3.7 Binding of Process Ports

In order to satisfy the interaction safety requirement, ports are further refined into exitports, and entryports.

An **exitport** specifies a service *required* by a component, for example, the process definition:

```
Task Module Pdef
        Exitport sin : real Reply real
        Exitport open : string Reply integer
End
```

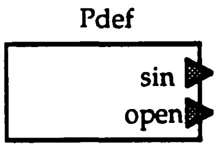


Fig 3.8 Process Exitports

specifies a process Pdef with requirements for services sin and open. Services model synchronous communication and are analogous to functions in procedural languages. Currently services are declared with a single argument called the request type, and a single result called the reply type.

port name : *request type* **reply** *reply type*

An **entryport** specifies a service *provided* by a component, for example, the process definition:

```
Task Module Qdef
        Entryport cos : real Reply real
        Entryport sin : real Reply real
        Entryport open : string Reply integer
        Entryport close : integer Reply integer
End
```

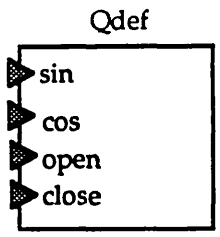


Fig 3.9 Process Entryports

specifies a process Qdef that provides services sin, cos, open and close.

Safe interaction binding is handled by allowing binding of compatible ports only. Ports are compatible only if one port is an exitport, the other an entryport, and the corresponding request and reply types are identical. For example, in the following

```
Use Pdef; Qdef;
Create P1 : Pdef; P2 : Pdef; Q : Qdef;
Link P1.sin to Q.sin
Link P1.open to Q.open
Link P2.sin to Q.cos
Link P2.open to Q.close
```

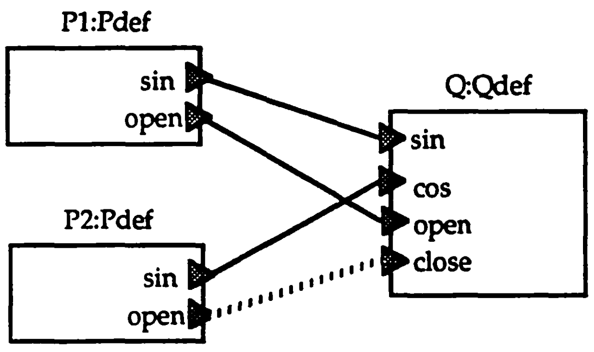


Fig 3.10 Bindings of Process Exitports to Process Entryports

the only illegal binding is the last one, where the request types are not the same. The binding of P2.sin to Q.cos is legal, since the safety rules have been met. The safety rule does not require port names to be identical as in CCS, nor that interaction safety is left to user-supplied processes to check as in HPC.

Ideally it would be desirable to attach the behaviour specification of services to ports, and match on some form of behavioural equivalence. This has, however been difficult to implement in practice, although systems such as Inscape [Perry87] show promise.

The division of ports into exitports (requirements) and entryports (provisions) [Tichy79, Wolf85] is crucial to ensure configuration-independent components. In languages without this division, the binding between modules is normally determined by explicit imports made in the the specification parts of modules. The binding forms an implicit dependency graph that constitutes the interaction structure of the program. This form of binding is inadequate since the relationships between modules are direct, e.g module A imports and calls service R from module B where module B, is named explicitly within module A. That is, the provider of a service R is named directly. Omitting the name of the provider and stating only the name of the service in the requirer would lead to modules that are loosely coupled, configuration independent and re-configurable.

In systems such as [Minsky83, Wolf85], modules can name not only the items to export, but also which modules can import them. This gives the module implementor greater control on the usage of modules but also leads to inflexible tightly coupled modules, that are less re-usable.

A more deductive approach is taken in [Levy84] where the requirements of a module are automatically determined by the programming language compiler in terms of the functions called but not defined within the module. A unifying algorithm is later invoked that attempts to find matching provisions for each requirement, based on the name and parameters of functions. Although the deduction of requirements and the use of unification for automatically generating the interaction structure of a program may be convenient it leads to a programming style, that fails to exploit the interaction structure of programs and which relies on a global name space for functions. Problems can occur if there are functions whose names and parameters clash.

3.3.1. Notify Ports

The configuration languages also supports the declaration of ports, called notify ports for defining asynchronous interactions. Notify ports are declared without reply parts, as in:

```
Task Module N
        Exitport xp : integer
        Entryport ep : real
```

End

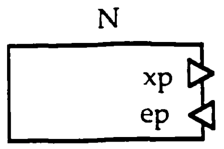


Fig 3.11 Process Notifyports

Notify exitports send out values only, while notify entryports receive values only. We call the argument type of a notify port a notify type. The binding rule for notify ports requires one port to be a notify exitport, the other to be a notify entryport and the notify types to be the same.

3.3.2. Fan-In, Fan-Out

The binding rules of our language do not preclude the binding of entryports to more than one exitport, or exitports to more than one entryport. Many-to-one bindings of the form:

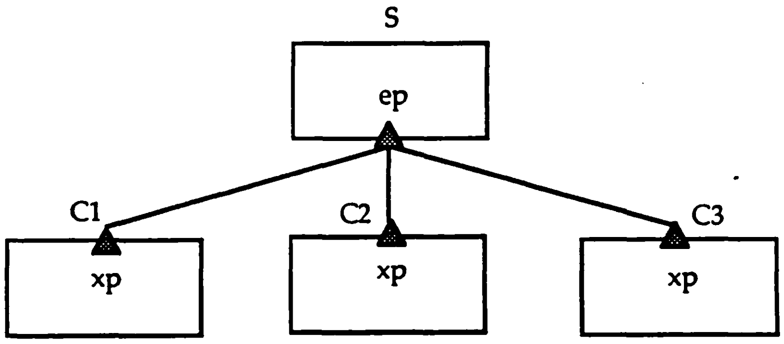


Fig 3.12 Fan-In

are allowed, as are one-to-many bindings of the form:

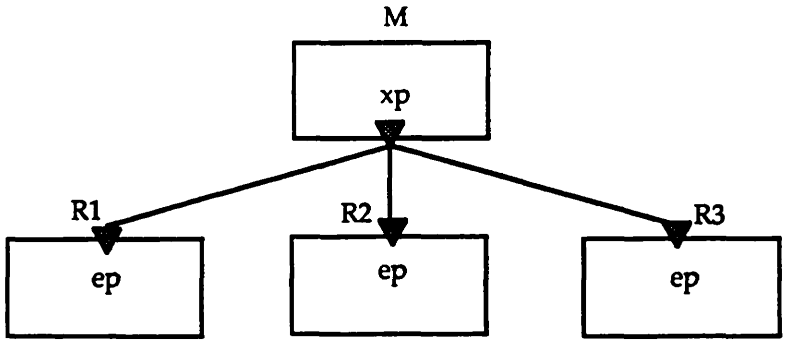


Fig 3.13 Fan-Out

Fan-in is useful for specifying server-client interactions, while fan-out is useful for multi-destination interactions. The number of entryports sent to, and the number of results returned by ports that fan-in and fan-out is programming language dependent.

3.3.3. Review

Given a structuring mechanism with processes, bindings and ports it is possible to build distributed programs simply, clearly and safely, but the mechanism is not scalable or modular. A mechanism for treating collections of processes as components is needed.

3.4. GROUP MODULES

Group modules (groups) are the means to specifying large structures in a modular way. Groups unify the concepts so far presented into a coherent and flexible structuring mechanism.

3.4.1. Encapsulation

Groups firstly provide an encapsulation mechanism for program structures. The following group definition encapsulates the three sub-components X1, X2 and X3:

```
Group Module Alpha
        Use P;
        Create X1 : P; X2 : P; X3 : P;
        Link X1.xp to X2.ep
        Link X2.xp to X3.ep
End
```

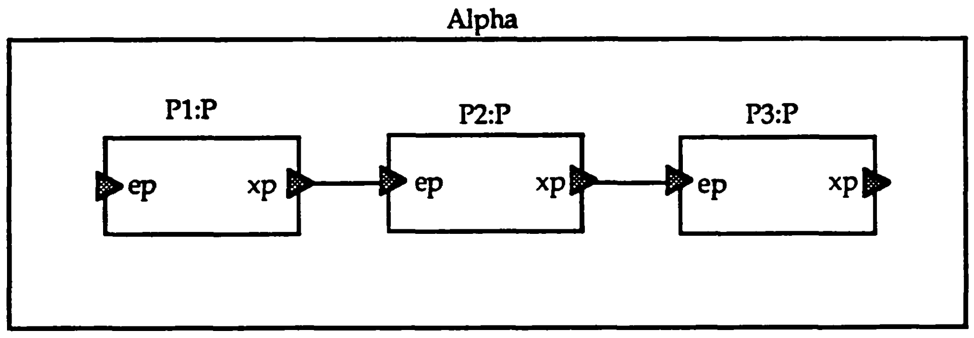


Fig 3.14 Example of a Group Module

Group definitions can be substituted for process definitions, for example in context definitions, and instantiation declarations. Like processes, groups can be multiply-instantiated, as in the following

```
Use Alpha;
Create A : Alpha; B : Alpha;
```

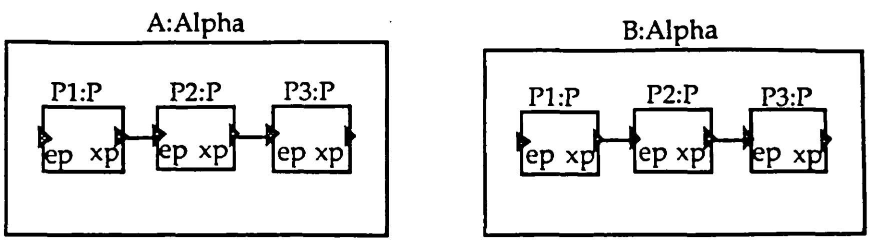


Fig 3.15 Multiple Instantiation of a Group Module

The uniform treatment of groups and processes, leads to a black-box approach to components in our language. It is not possible to distinguish between components that are atomic (processes) and components that are compound (groups). Processes can be replaced by groups and vice-versa.

Since groups can be instantiated in other groups, groups support hierarchic program structuring better than modular programming languages which disallow modules of modules or multiple instantiation of a single module.

3.4.2. Hierarchic Binding

Groups also support structural abstraction and modular decomposition. This is achieved with the same mechanism used for processes, namely ports. Group ports are declared in a similar way to process ports.

```
Group Module ....
        Exitport .....
        Entryport .....
        <rest>
End
```

Only the name of the group, and the names of ports are visible outside the group definition. The correspondence between group ports and instances declared within the group is specified by hierarchic binding declarations of the form:

Inbound Hierarchic Binding:

Link *groupEntryport* **to** *InstanceEntryport*

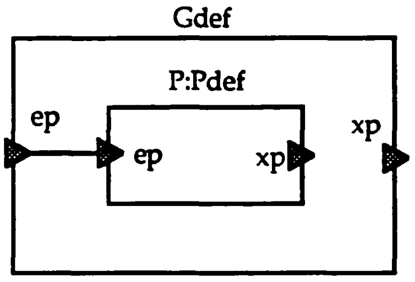


Fig 3.16 Inbound Hierarchic Binding

Outbound Hierarchic Binding:

Link *InstanceExitport* **to** *GroupExitport*

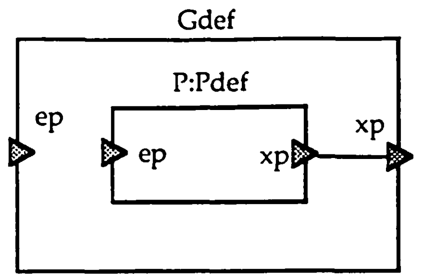


Fig 3.17 Outbound Hierarchic Binding

The first form exports out a group provision of a sub-instance, the second form exports out a group requirement of a sub-instance. An additional binding rule is also allowed:

Forwarded Binding:

Link *groupEntryport* **to** *GroupExitport*

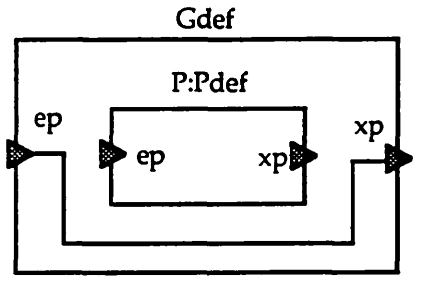


Fig 3.18 Forwarded Hierarchic Binding

This defers the provision of a service by forwarding it back out of the group as a requirement, ie. this states that the group cannot fulfil the service, but wishes to export it out as a requirement. In fact forwarded binding is useful rule as it allows the declaration group modules that act as *switching structures,* for example the group:

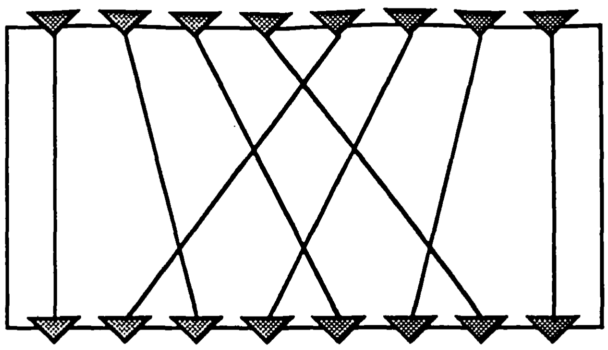


Fig 3.19 Example of a Perfect Shuffle Switching Structure

is interconnected as a perfect shuffle [Stone71]. Another use for forwarded binding is in the declaration local logical buses. These comprise a single group module with a forwarded binding as in:

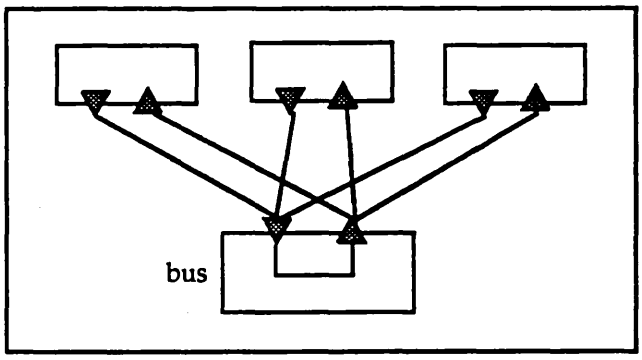


Fig 3.20 Example of a Logical Bus

Components binding to the logical bus do not have to specify and therefore know the number or names of their bound counterparts. Forwarded binding also displays the transitive property:

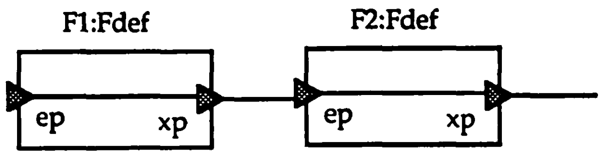


Fig 3.21 Transitive Bindings

3.4.3. Inheritance (Incremental Structuring)

Inheritance allows new components to be constructed from old components, new services to added or existing services to be changed. Inheritance-based languages are thus suited to programming applications in an incremental fashion. The grouping mechanism is sufficiently powerful to construct components by defining the differences. The following examples illustrate this.

A new component can be defined as an incremental additional to an existing one, as in:

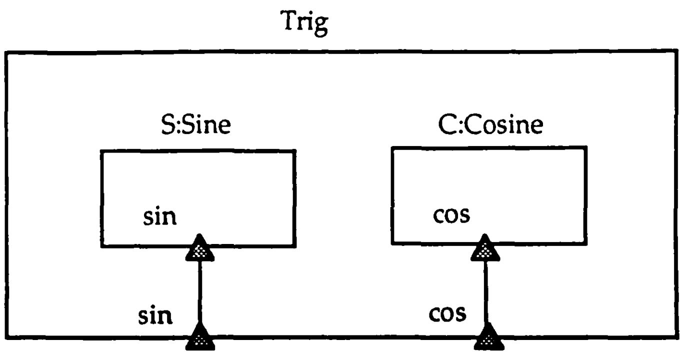


Fig 3.22 Incremental Structuring of Components

New components can be constructed from several existing components also:

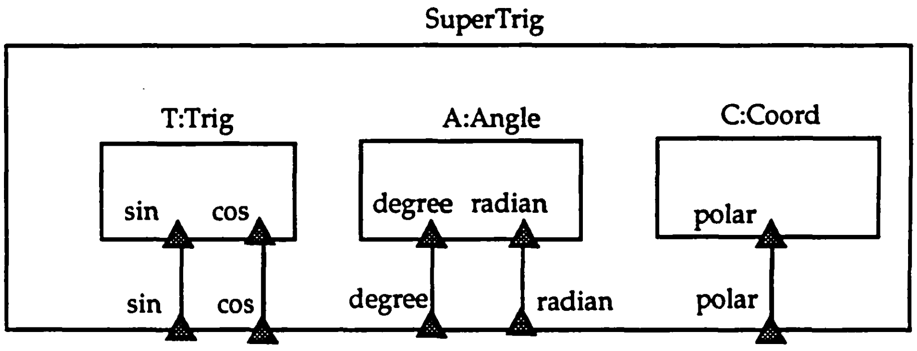


Fig 3.23 Incremental Structuring from Several Components

Conflicts are handled by renaming, since name overloading is not supported.

It is important to note that the atomic components of groups are concurrently active processes, whereas most inheritance languages support passive class hierarchies. In our groups, component requirements can be inherited, whereas in inherited languages only component provisions in the form of methods are inherited.

A new component can be defined that overrides an existing component:

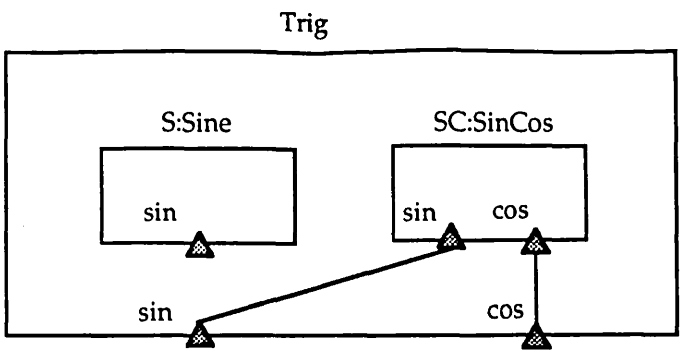


Fig 3.24 Overriding Components

3.4.4. Coalescing. Multicasting and Loop Backs

Other forms of group binding are also supported in the language including coalescing, multicasting, and loop backs.

Coalescing (Multiplexing)

Several sub-component requirements can be bound to one requirement:

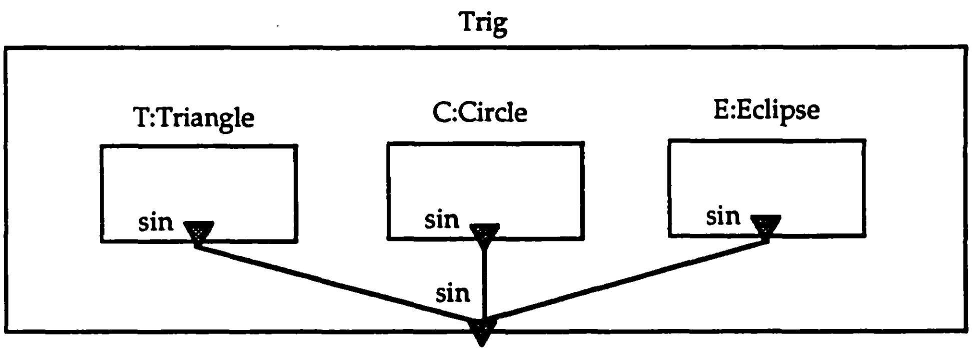


Fig 3.25 Hierarchic Coalescing

Multicasting

A single service provision can be bound to several sub-component provisions:

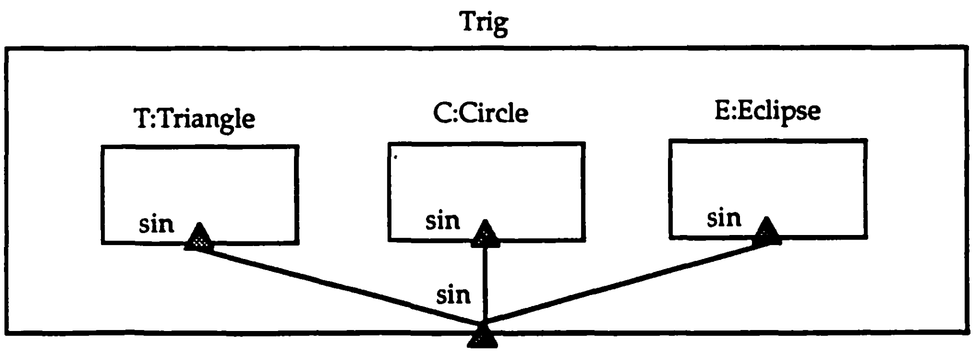


Fig 3.26 Hierarchic Multicasting

Multicasting could be treated as a form of overloading, with one or more of the bound sub-components responding to the service with responses being based on the contents of the message. This semantic would be better suited to configuration languages with untyped or polymorphic port types.

Loop Backs

The rules for component binding allow group entryports to be bound to group exitports. This rule allows for a component exitport can be bound back to a entryport of the same component. This allows the configuration programmer to either provide his own service, or use a service provided by the component (see below), or ignore the requirement.

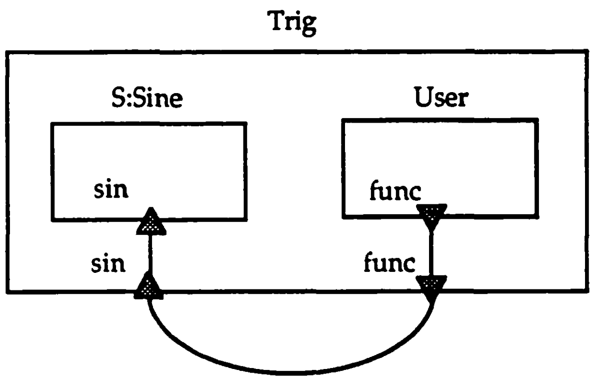


Fig 3.27 Loop Back Binding

A possibility exists for establishing circular binding, such as:

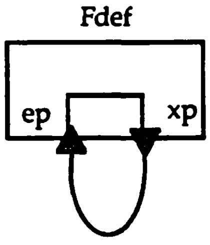


Fig 3.28 Circular Binding

Circular bindings are currently considered as null bindings and silently ignored. They should probably be made illegal.

3.4.5. Binding Rules

The binding rules for the configuration language are presented below. Geometrical representations are used to simplify the definitions.

A component (group or process) is represented by a quadrilateral:

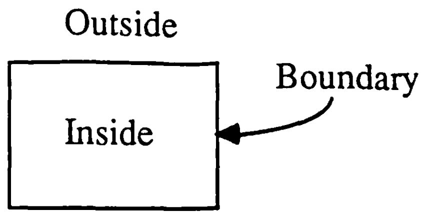
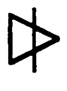
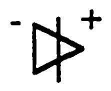


Fig 3.29 Geometric Representation of a Component

Components can nest. Boundaries cannot intersect. A port is represented by a triangle cutting across a component boundary like  (i.e. not bisecting an angle). The side of the triangle with one angle is labelled '+'. The side of the triangle with two angles is labelled '-', ie .

An exitport is represented by a port with the '+' side of the port on the outside of the component boundary:

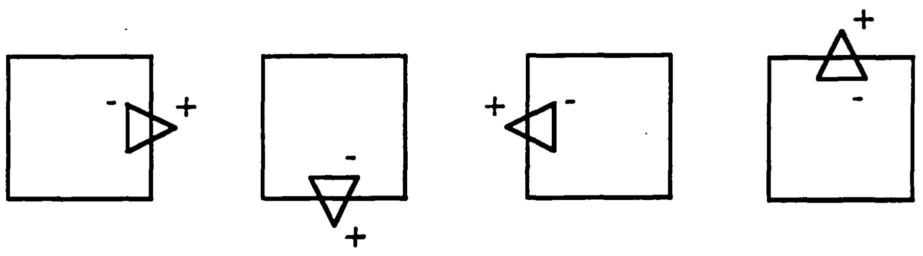


Fig 3.30 Geometric Representation of an Exitport

An entryport is represented by a port with the '-' side of the port on the outside of the component boundary:

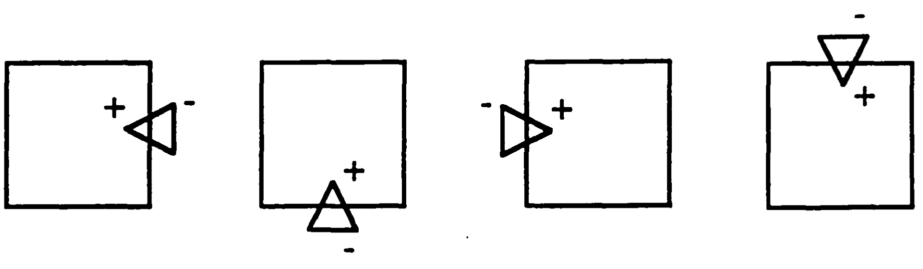


Fig 3.31 Geometric Representation of an Entryport

Given the definitions above, a binding is represented by a line from the '+' side of one port to the '-' side of another port. Bindings are not allowed to cross component boundaries. Ports can be bound to several other ports. The first configuration below is legal, the second is illegal.

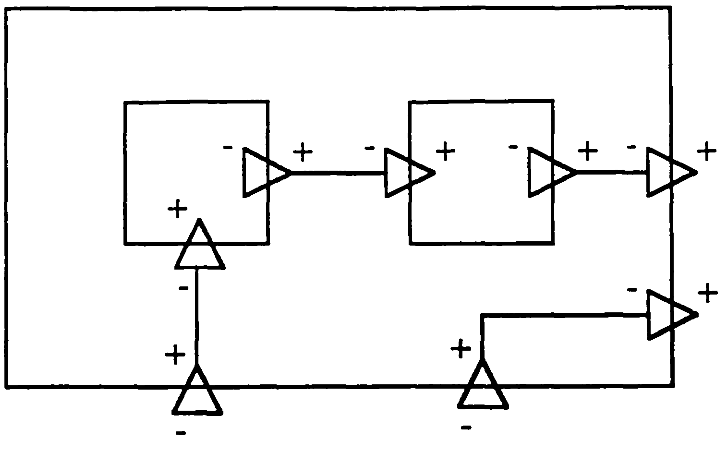


Fig 3.32 Legal Bindings

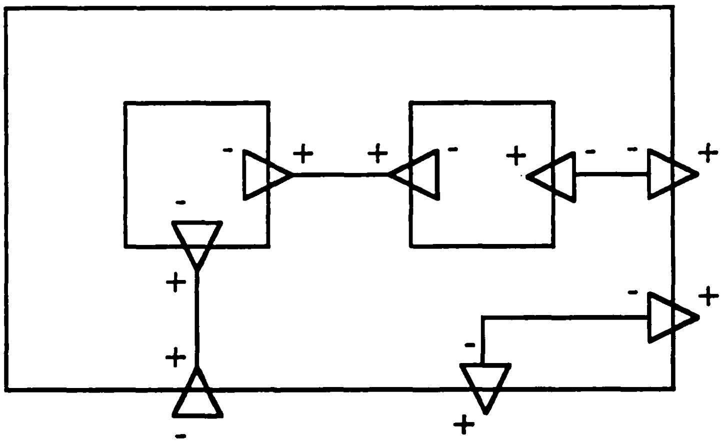


Fig 3.33 Illegal Bindings

Interaction safety is ensured by labelling ports with a tuple representing their port types, either (requestType, replyType) or (notifyType). Bindings are type-safe if the tuples of the ports at each end of a binding line are 'compatible'. Currently compatibility is defined if tuples have the same cardinality, and pair-wise elements of the tuple are type equivalent.

3.4.6. Final Comment

Groups are a powerful mechanism for structuring large distributed programs in a modular way. In order to increase the expressiveness of our language we need to add mechanisms for parameterisation, replication, variation and recursion.

3.5. PARAMETERISATION

Parameters are used to control the size and topology of configuration structures and also to supply initial values to atomic components. Group parameter declarations are written in the style of Pascal value parameter declarations[2], for example:

```
Group Module A (a:integer; ch:char)
```

Given a parameterised component definition, actual parameters can be supplied during component instantiation, for example

```
Use thermometer;
Create celsius : thermometer (45,'C')
Create fahrenheit : thermometer (93,'F')
```

Default parameterisation simplifies component instantiation further and is also supported, for example:

```
Group Module thermometer (initial:integer=98; unit:char='F')
```

3.6. REPLICATION

Replicated or array structures occur so frequently in practice that their inclusion in the configuration language is almost obligatory.

Replication occurs in three places in the configuration language, replicated component instances, replicated ports, and replicated binding. The mechanism used to define replicated structures is universal quantification. Each replicated declaration is prefixed by one or more quantifiers of the form:

ForAll *BoundIdentifier* : [*LowValue* .. *HighValue*]

where *BoundIdentifier* takes on successive values starting from *LowValue* and ending with *HighValue*. LowValue and HighValues are expressions[3] of integer, char, or boolean type. LowValue and HighValue must be of the same type. Group parameters can be used in bound expressions to control the size and topology of replicated structures. For historical reasons the keyword **Family** can be used in place of the keyword **ForAll**.

Replicated component instances are declared by suffixing the instance name with the bound identifier(s) specified in square brackets, for example:

```
Create ForAll I : [1..10]
        A[I] : At (I)
```

2 Only value parameters of the simple types integer, char, boolean, real and string are currently allowed.

3 Expressions are the simple expressions of Pascal, except that it is possible to call imported functions, provided they do not reference non-local data.

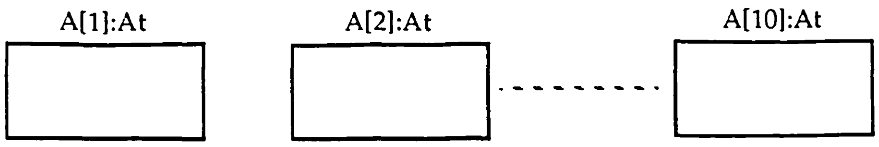


Fig 3.34 Replicated Instantiation

Note the passing of the bound identifier I as a parameter to individual replicated instances. Multi-dimensional component instances[4] can be declared by supplying more quantifiers , for example:

```
Create ForAll I : [ 1..10 ], K : [ 1..8 ]
        A[I,K] : At (I,K)
```

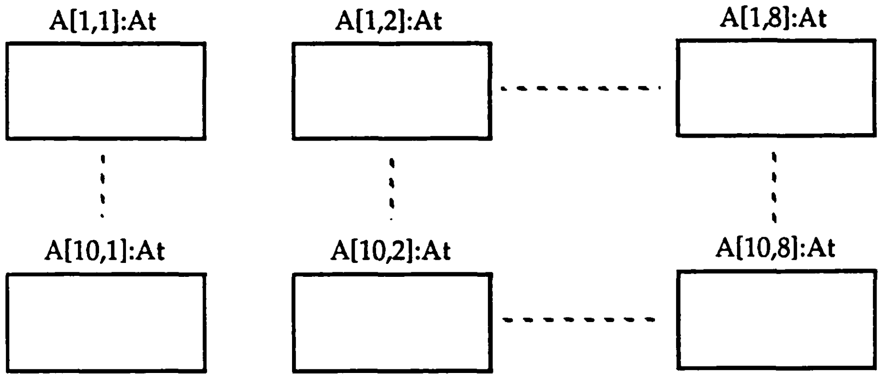


Fig 3.35 Two-dimensional Replicated Instantiation

Replicated ports can be declared analogously to replicated component instances, for example:

```
Entryport ForAll I : [ 10..15 ]
        func [I] : integer Reply char
Exitport ForAll I : [ 10..15 ]
        files [I] : string Reply integer
```

If the bound identifier(s) are not actually used in the declaration then short forms of the replicated component instantiation, replicated port declarations can also be used. These replace the bound identifier in the declaration by the desired lower and upper bounds, for example:

```
Create A[1..10] : At
Entryport func [10..15] : int Reply char
Exitport files [10..15] : string Reply int
```

Quantifiers are also needed for the binding declaration, to allow replicated component instances and replicated ports to be bound:

```
Link ForAll K : [ 0..N ]
        A[K].xp to filesXp [N-K]
```

4 Multi-dimensional replicated structures are not currently implemented.

Replicated ports and replicated component instances are selected by supplying an indexing expression in square brackets after the replicated port/instance name. This expression can include group parameters as well as bound identifiers. Quantified binding declarations often take more than one quantifier, for example,

```
Link ForAll I : [ 1..10 ], K : [ 1..20 ]
        A[I].xp[K] to P.ep[K];
```

Distributed programs with quite complex interaction topologies can be expressed using such quantifiers, e.g. pipes, rings, grids, hypercubes, butterflies, switching networks and tree topologies.

3.7. VARIATION

It is often convenient to allow a group to have the potential to define more than configuration structure, and to be able to defer the choice until the group is instantiated. This can be achieved by allowing component instantiation and component binding declarations to be prefixed by a boolean guard.

```
When Guard  Create .....
When Guard  Link ...
```

Typically the guard includes one or more group parameters which control the form of variant structure produced.

Guards are disallowed for port declarations. This is to ensure that groups have fixed interfaces. A more fluid configuration language could include this as a possible extension.

3.8. RECURSION

A structure is said to be recursive if it is defined in terms of itself. Recursion is a powerful program structuring mechanism for distributed programs. The characteristic feature of recursive structures is their ability to vary in size.

In the configuration language group names can be used within their own definition, for example:

```
Group Module a
        Create aa : a;
        Create b;
End
```

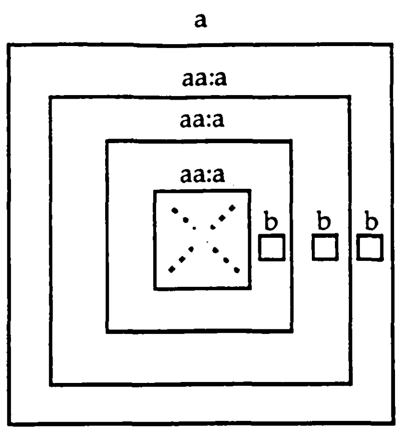


Fig 3.36 Recursive Structuring

This particular example defines an infinite distributed program configuration, and is not useful in practice, unless a lazy evaluation scheme is used for elaborating configuration specifications. In order to limit recursion, guards and parameterisation can be used, for example:

```
Group Module a (k:integer)
        When k>0 Create aa : a (k-1);
        When k>0 Create b;
End
```

In this example it is worth noting the existence of an empty group instance in the limiting case.

3.9. DYNAMIC MODIFICATION

In order to perform dynamic modifications to the structure of a running program, declarations are also needed to remove instances and bindings. The form of these declarations[5] is simply:

Remove *Instance*

Unlink *port* **from** *port*

3.10. INTERACTION SAFETY

The request, reply and notify types that can be specified for a port are not restricted to simple types such as integer, real, and character. The configuration language supports a much richer collection of types. The typing language used to describe port types is the type definition sublanguage of Pascal, enhanced with the type extension mechanism of Oberon [Wirth88a, Wirth88b]

5 Currently these declarations only be applied dynamically to the top-level structure of a distributed program.

Many distributed programming systems are typeless, or provide a poor selection of types. Pascal was chosen because it has a rich set of types, and a type checking philosophy that values secure programming. Typeless programs are often harder to debug, while type-poor programs often require greater effort by the programmer.

In addition to using Pascal types for port types, the configuration language has also adopted the strong typing rules of Pascal for parameter conformance and binding compatibility.

In fact port definition can be considered an issue orthogonal to the configuration languages, other typing systems or interface definition languages [Hayes87] could be employed without adversely affecting the configuration language.

3.10.1. Port Types

The port types supported by the configuration language are the standard types integer, real, char, and boolean, plus the types byte (a subrange of 0 to 255), natural (an unsigned integer), longint, and signaltype (for void values). Enumerated, subrange, array, record, and set types are also supported, as are packed types. The remaining types, pointer types, variant record types, and file types are either disallowed or supported to a lesser degree.

Pointer types and types with embedded pointer types need to be disallowed, as they imply the possibility of non-exclusive access to data values by concurrent processes. Pointer passing between distributed or heterogeneous address spaces is also problematic. A pointer passing semantic that requires complete copying of the entire pointed at heap or a semantic that requires backward communication to the originating processor to dereference pointers are possible remedies, although ones with a high implementation overhead.

For efficiency reasons, pointer types are currently allowed as message types for bound processes which reside within a shared address space. This concession allows two or more processes to reference a common (and usually large) data structure. Mutual exclusion is left to the programmer to ensure.

Untagged variant records are disallowed as port types since it not possible to know which variants in a value are current.

File types and types with embedded file types are prohibited since Pascal defines no semantic for file assignment.

3.10.2. Port type compatibility

The compatibility rule used is Pascal type equivalence, not Pascal assignment compatibility. Assignment compatibility complicates the semantics by requiring coercion considerations to be addressed, particularly in hierarchical bound ports that are multi-cast or coalesced. For example, if a group exitport of type real, is bound hierarchically to local a exitport of type integer, and also to a local exitport of type real, is the binding allowed? Similar considerations apply for subrange types.

The use of a type equivalence rule for component binding greatly simplifies the understandability of the configuration language without unduly affecting expressibility.

3.10.3. Extended Message Types

The type language also includes the type extension mechanism of Oberon [Wirth88a, Wirth88b]. This allows record types to be defined incrementally, and the strict type equivalence rule to be relaxed to a subtype equivalent rule. Extended types are similar to the class concept found in object-oriented languages, although in the framework of Pascal, procedures and functions are not types, and so cannot be fields of records. The following example illustrates their declaration:

```
type    A =    record
                        a1 : integer;
               end;
        B =    record (A)
                        b1 : real;
               end;
        C =    record (B)
                        c1: packed array [1..10] of char;
               end;
        D =    record (A)
                        d1, d2 : char;
               end;
```

The type hierarchy can be drawn as follows:

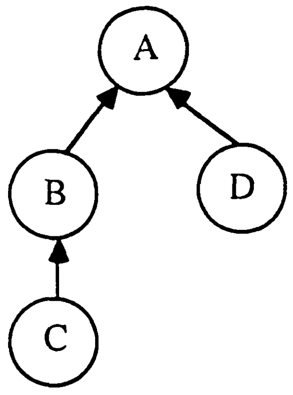


Fig 3.37 Example of a Type Hierarchy

Types B and D are extensions (or subtypes) of type A. Type C is an extension (or subtype) of type B and also of A. A is a supertype of types B, C, and D. B is a supertype of C. Thus type B has fields a1 and b1, type C has fields a1, b1 and c1. Type D has fields a1, d1, and d2.

A record type may have many subtypes but only one supertype. Extension of variant records is not allowed. Extending a non-variant record to have variants is allowed, but makes the extended record non-extendable.

3.10.4. Extended type compatibility

The type equivalence binding rule for ports of extended types can be relaxed to subtype compatibility in the following way: allow an outgoing port to be bound to incoming port if the outgoing port's request type is equivalent to, or an extension of the incoming port's request type, and if the incoming port's reply type is equivalent to, or an extension of the outgoing port's reply type. More concisely, if given a binding

$$\begin{tikzcd}\textbf{Outgoing Port} & \textbf{Incoming Port} \\ \text{RequestA Reply ReplyA} \arrow[r] & \text{RequestB Reply ReplyB}\end{tikzcd}$$

then the binding is subtype compatible if

(RequestA >= RequestB) and (ReplyB >= ReplyA)

where >= is the type relation "is-equivalent-to-or-an-extension-of"

As in Oberon, the type extension mechanism is applicable to types that are pointers to extended records.

3.10.5. Definition Modules

In order to modularise and make orthogonal the declaration and use of port types a special component called a definition module is provided. Definition modules serve to define the port types of a distributed program in a configuration language and programming language independent manner. The following example illustrates their syntax and use.

```
define M: a, b, c, z ;                        {definition M exports a,b,c and z }

use    A: x, y;                               {import x and y from definition module A}
       B: z<y>;                               {import y from definition B, but rename to z}

type   a=array [1..10] of x;
       w=array [y] of real;                   {w is not exported}
       b=record ch:char; pin:w end;
       c=set of char;
end.
```

Types to be exported "outside" the definition module are listed after the definition module name. Components may use the type definitions exported from a definition module by a context definition, which serves to make the types listed after the definition module name, known within the scope of the using component. Renaming can be used to distinguish two imported types with the same name.

Definition modules allow the message types for a distributed program to be consistently used by several components without resorting to re-definition in each component. In comparison, include files suffer from the lack of visibility controls on types, and lead to unnecessary processing of redundant types.

3.11. CHAPTER SUMMARY

This chapter has presented a declarative configuration language that fulfils the requirements for structuring distributed programs identified in Chapter 2. A single powerful structuring mechanism called a group module forms the basis of the language.

Group modules are based on a minimal number of concepts, atomic processes, ports, instances and bindings. Processes are the unit of concurrency and programming in our model. Processes can be programmed in any suitable programming language. Groups are the unit of configuration and can encapsulate both processes and subgroups

A uniform approach is taken in the treatment of groups and processes. Both define a component type that can be used to instantiate components at higher levels of configuration. This enables the specification of arbitrarily large program hierarchies.

Component types are defined solely in terms of a port interface that abstracts and hides the internal details of the component, thus promoting modularity. Ports are declared within a component definition and define the interaction points of the component. Ports are thus the only 'gateways' into and out of a component. Ports are directed and define either a service requirement (entryports) or a service provision (exitports). The inclusion of the service requirements of a component in the interface of components leads to loosely coupled components and components that can be reconfigured. Bindings establish the interaction topology of a distributed program. Binding declarations require two ports, one providing a service and other requiring the service. Bindings can be one-to-one, one-to-many, and many-to-one.

Interaction safety is ensured by typing ports and requiring bound ports to be type compatible. A rich selection of types is also supported.

The provision of quantifiers, guards and parameters enable replicated, variant, and recursive structures to be expressed simply and clearly. This yields a very powerful configuration language.

The next chapter demonstrates the power and applicability of the language with a concrete set of examples. Chapter 5, 6 and 7 show how configuration specifications can be compiled and efficiently executed in a distributed environment.

Chapter Four Examples

This chapter demonstrates the power and applicability of the configuration language with a series of graduated examples. The first example presents the structure of a simple flow communication subsystem and is used to introduce the definition, instantiation and interconnection of components. The second example presents the structure of a distributed solution to the dining philosophers problem and demonstrates the application of replicated component instances. The third example presents the structure of a distributed run-time executive and demonstrates the use hierarchic structuring. The final example presents the structure of parallel solution to Batcher's bitonic sorting algorithm and demonstrates the application of recursive configuration structures.

4.1. SIMPLE FLOW CONTROL PROTOCOL

This first example illustrates simple configuration instantiation and interconnection of a small program. The program simulates a flow control communication subsystem between a producer process and consumer process. The structure of the simulation pipeline is shown below:

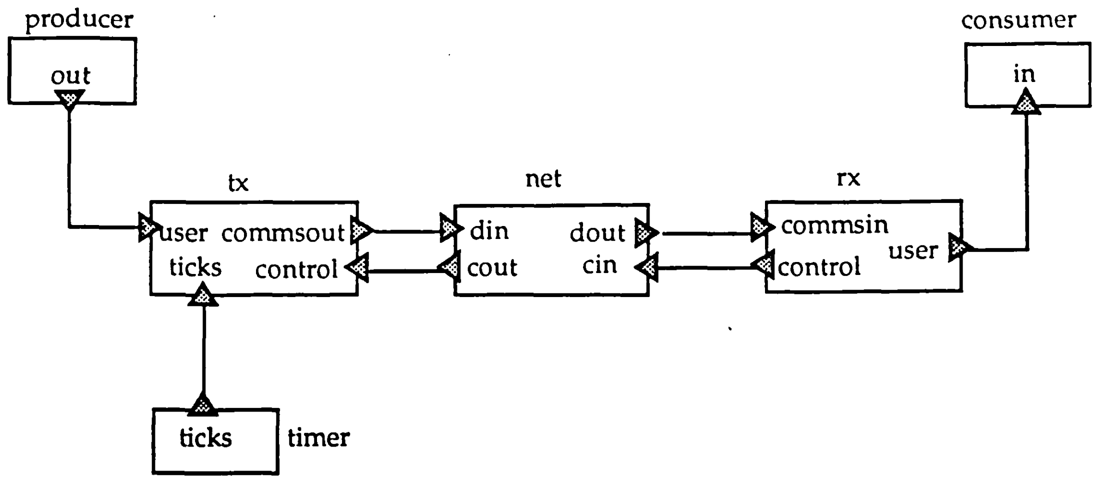


Fig 4.1 Flow Control Protocol Example

The simulation is implemented by the tx, timer and rx modules which provide the implementations for error and flow control, and the net module which simulates a physical network that may lose, corrupt or duplicate messages. The structure shown can be described with the configuration description:

```
group module flowcontrol (messages:integer;  bias:integer=5);

use     consumer;  producer;              sender;        receiver;
        net;       timer;

create  consumer;  producer (messages);   tx:sender ;    rx:receiver;
        net (bias);  timer;

link    producer.out       to tx.user;
        timer.ticks        to tx.ticks;
        net.cout           to tx.control;
        tx.commsout        to net.din;
        net.dout           to rx.commsin;
        rx.control         to net.cin;
        rx.user            to consumer.in;
end.
```

Apart from some parameterisation the description is a straightforward encoding of the graphical form. The **use** clause selects the component types required, the **create** clause instantiates a single instance of each, and the **link** clause interconnects the instances in the required topology. The example illustrates a flat program structure, equivalent in power to a programming language with a import-export modular structure, such as Modula2, although in our language the bindings are separated from the module, and parameterisation of modules is allowed.

Note that the configuration description does not indicate whether a module is a primitive process module or a configuration module, nor does it indicate the types or directionality of bound ports. This is also reflected in the graphical description which for space restrictions also omits module type names and parameter information of components.

The configuration header for flowcontrol illustrates the definition of both a mandatory parameter (*messages*) and optional parameters (*bias*). The first is passed down to the producer module, the second to the net module. Apart from the configuration header, other configuration declarations can be specified in any order.

It is worth noting that this example is a little artificial since for 'real' systems the communication structure would probably be abstracted into a separate group module as in:

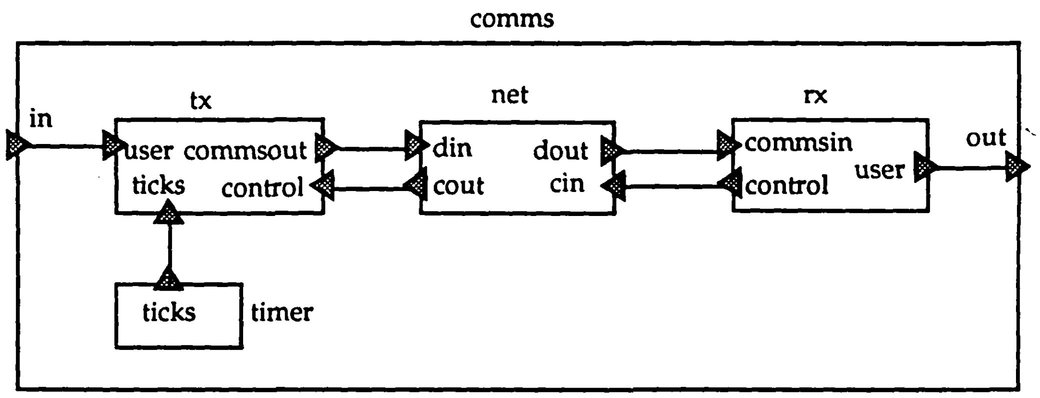


Fig 4.2 Flow Control Protocol Group Module

The use of hierarchic structuring is considered further in the third example.

4.2. DINING PHILOSOPHERS

The second example introduces replicated component instantiation and binding. The program provides a solution to the classic dining philosophers problem [Dijkstra68] devised for assessing the capabilities[6] of synchronization primitives.

The problem involves five philosophers seated around a table. In the middle of the table is a bowl with an infinite supply of spaghetti. Half way between each philosopher is a single fork. Philosophers spend their time moving from a thinking to hungry to eating and back to thinking state. A philosopher must hold both adjacent forks in order to eat spaghetti from the bowl. The circular structure implied by the problem of philosophers and forks around a table can be used to obtain a simple solution to this problem. For 4 philosophers we would require

6 Prohibit starvation, free from deadlock and maximise parallelism.

the following configuration structure:

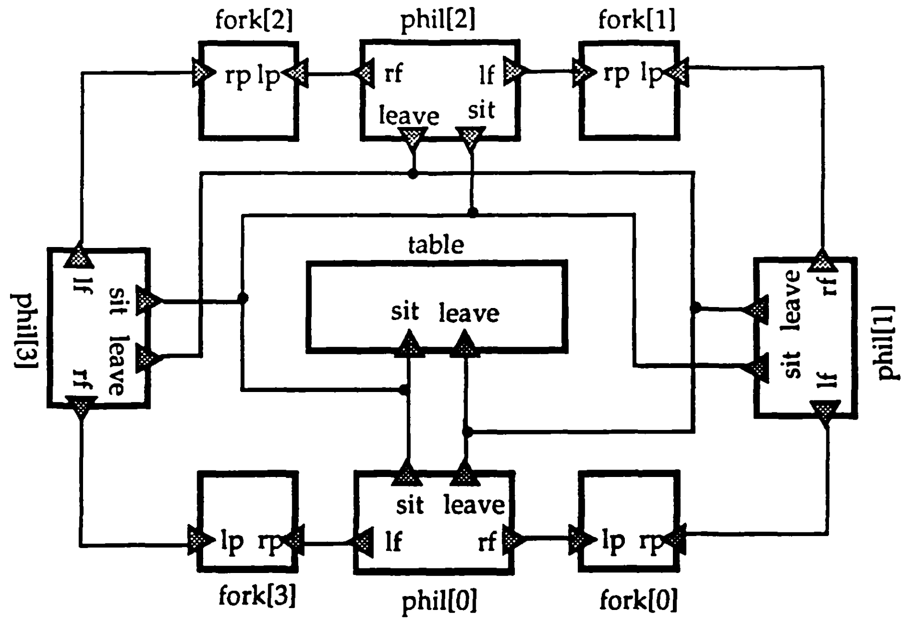


Fig 4.3 Dining Philosophers Example

A configuration description for this structure is given below where lp=leftphil, rp=rightphil, lf=leftfork, rf=rightfork:

```
group module diners (n:integer=4);
-- assert n >=2

-- dining philosopher modules
use     table;      phil;           fork;

create  table(n);

create  forall k:[0..n-1]
        phil[k] : phil(thinktime=2000,eattime=2000);
        fork[k]  : fork;

link    forall k:[0..n-1]
        phil[k].sittable        to table.sit;
        phil[k].leavetable      to table.leave;

        phil[k].rightfork       to fork[k].leftphil;
        phil[(k+1) mod n].leftfork
                                to fork[k].rightphil;

-- create and bind display windows for program
use     windman;

create  forall k:[0..n-1]
        pwin[k] : windman (0,(k+1)*14-13,(k+1)*14-5,8,17);
        fwin[k]  : windman (0,(k+1)*14-5,(k+1)*14+1,8,17);

link    forall k:[0..n-1]
```

```
                phil[k].std_write      to pwin[k].window;
                fork[k].std_write      to fwin[k].window;

    end.
```

In the description parameterisation is used to dimension the number of philosophers and forks, and to pass to the philosopher processes the length of the periods for thinking and eating. The **forall** clauses act as universal quantifiers over the **create** and **link** declarations and range from 0 to the number of philosophers minus 1. The bound identifier k is used in expressions to link the philosophers and forks in a ring, as well as in expressions for calculating the display coordinates for a set of window modules. The window modules are declared for each philosopher and fork and are used to animate the output of the program. For 4 philosophers the configuration structure with the window modules is:

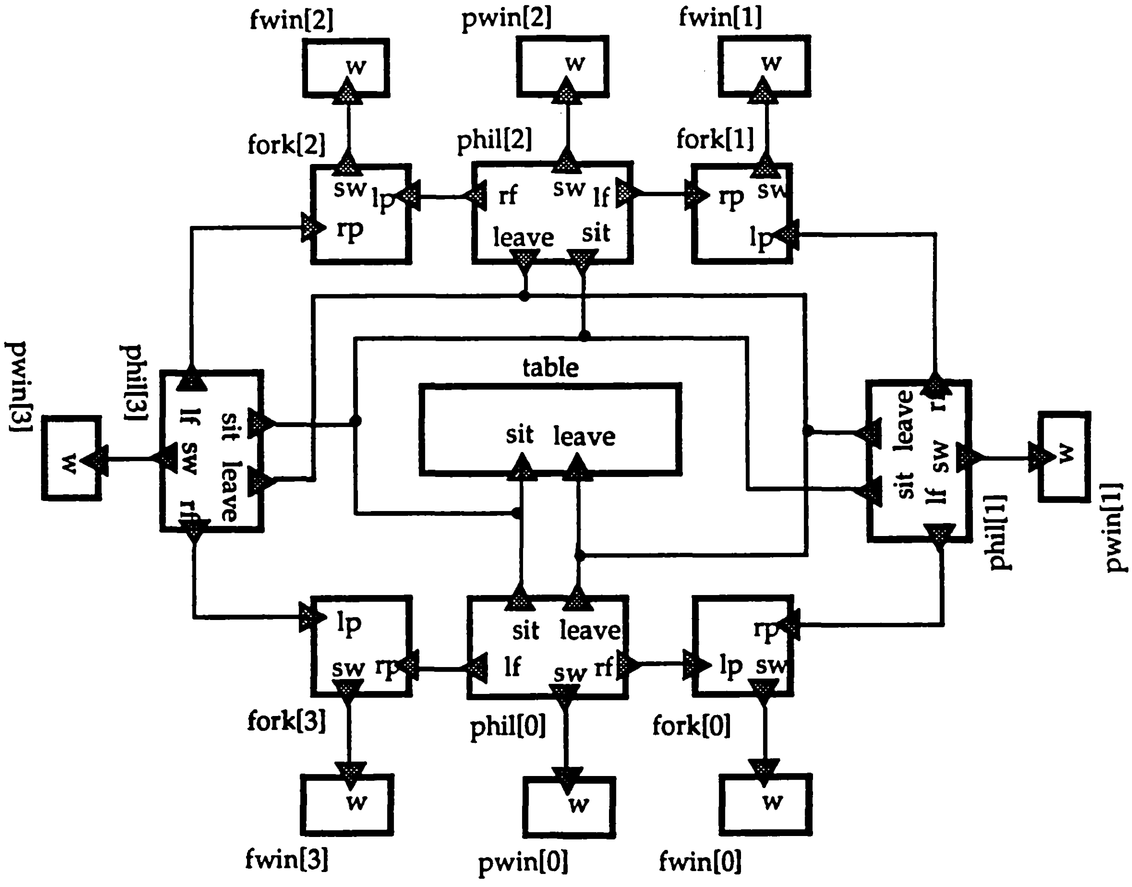


Fig 4.4 Dining Philosophers Example with Window Components

where lp=leftphil, rp=rightphil, lf=leftfork, rf=rightfork, w=window, and sw=stdwrite.

The configuration also makes use of bindings that fan-in into the table entryports. A more modular solution to this problem would group together a philosopher, a fork and two windows into a single group module, and perform circular binding of an array of such groups.

4.3. RUN-TIME EXECUTIVE

The third example demonstrates the use of group modules for hierarchically structuring programs. The example presents the overall structure of a Conic run-time executive and is described more fully in [Magee86]. In Conic each distributable part of a program requires an instance of such an executive in order to enable that part for execution. The executive described enables execution under Unix®. The structure of the executive is shown below

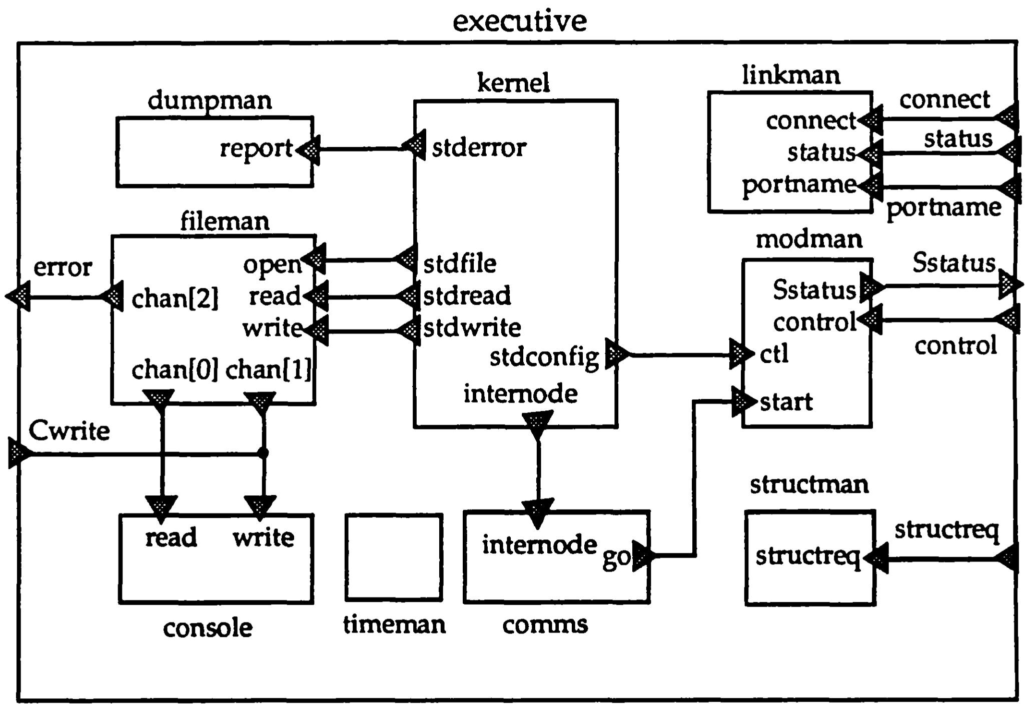


Fig 4.5 Executive Group Module

These modules provide various services for distributable program parts including: local process creation, scheduling, and interaction (kernel), reporting process crashes (dumpman), file I/O (fileman), terminal I/O (console), remote process interaction (comms), dynamic reconfiguration (linkman and modman), configuration structure inspection (structman), and a clock timer (timeman),

In Conic, every process component is supplied with a set of standard ports: a *stderror* for reporting errors in the process, ports *stdfile, stdread, stdwrite* for

® Unix is a registered trademark of AT&T.

performing I/O and a *stdconfig* port for requesting dynamic configuration events. In order to reduce the profusion of linking that would be required to link these ports for each process, the Conic kernel implements a default linking rule for these ports. Each standard port that is not explicitly linked is automatically linked to whichever ports the corresponding kernel standard port is linked to e.g. for the kernel above every unlinked stdread port would be linked to fileman.read. Again for simplicity the standard ports of processes have been omitted from diagrams.

The interface to the executive consists of a port for forwarding error output (error), a port for directly writing to the console (Cwrite), a port for access the structural topology of a distributable part (structreq), and a set of ports for performing dynamic reconfiguration operations (connect, status, portname, Sstatus and control)

The configuration description for this executive is given below:

```
group module executive (ticktime:natural=1000);

-- basic system modules
use       kernel;     timeman;              dumpman;     fileman;
create    kernel;     timeman(ticktime);    dumpman;     fileman;

-- configuration management modules

use       manage: connectT,linkstatusT,controlT,sifrecT;
          serverdefs:status_rec;

use       linkman;    modman;
create    linkman;    modman;

exitport  Sstatus : status_rec;
entryport connect : connectT reply signaltype;
          status : signaltype reply linkstatusT;
          portname : integer reply sifrecT;
          control : controlT reply status_rec;

link      connect           to linkman.connect;
          status            to linkman.status;
          portname          to linkman.portname;
          control           to modman.control;
          modman.Sstatus    to Sstatus;

-- Internode Communication subsystem

use       comms;
create    comms;

link      kernel.internode  to comms.internode;
          comms.go          to modman.start;
```

```
-- Console subsystem
use        console;
create     console;

use        types: rw_req;
exitport   error  : rw_req reply integer;
entryport  Cwrite : rw_req reply integer;

link       fileman.chan[0]    to console.read;
           fileman.chan[1]    to console.write;
           fileman.chan[2]    to error;
           Cwrite             to console.write;

-- Structure Query Handling

use        structure:structureB;
           structman;
create     structman;

use        structure:structureB;
entryport  structreq: integer reply structureB;
link       structreq          to structman.structreq;

-- Define default links for tasks --

link       kernel.std_config  to modman.ctl;
           kernel.std_file    to fileman.open;
           kernel.std_write   to fileman.write;
           kernel.std_read    to fileman.read;
           kernel.std_error   to dumpman.report;
end.
```

The configuration illustrates the form of declaration of interfaces, in particular the requirement to use context definitions to import port types. Two of the instances declared within this executive, console and comms, are in fact group modules with an internal substructure.

The console consists of a terminal driver that performs asynchronous read and write services for processes. The terminal driver also handles various Unix signals [Leffler89], such as input available (SIGIO), stop output (SIGTSTP), continue output (SIGCONT). These signals are caught by separate signal handler processes and passed to the terminal driver. The structure of the console is:

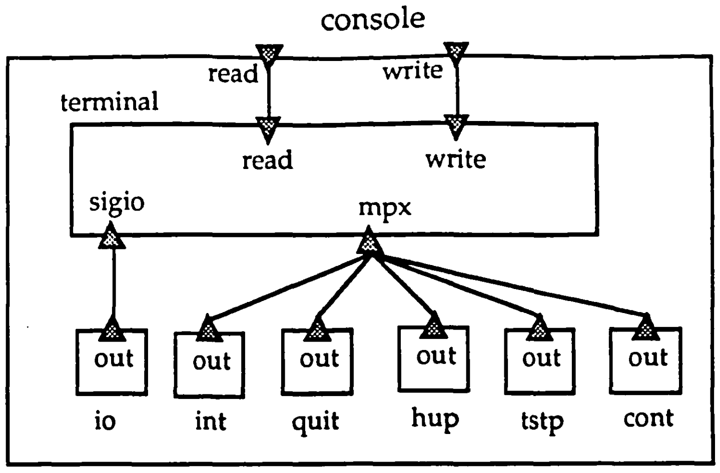


Fig 4.6 Console Group Module

The configuration description being:

```
group module console (ttymode:integer=0);

use        types: rw_req;
entryport  read, write : rw_req reply integer;

use        terminal;
create     terminal (ttymode);

link       read              to terminal.read;
           write             to terminal.write;

– Unix Signal handlers
use        signals:SIGIO,SIGINT,SIGQUIT,SIGHUP,SIGTSTP,SIGCONT;
           handler;

create     io:handler (SIGIO);
           int:handler (SIGINT);
           quit:handler (SIGQUIT);}
           hup:handler (SIGHUP);
           tstp:handler (SIGTSTP);
           cont:handler (SIGCONT);

link       io.out  to terminal.sigio;
           int.out, quit.out, hup.out, tstp.out, cont.out  to terminal.mpx
end.
```

The comms group consists of a socket driver that utilises the socket facilities of BSD Unix [Leffler89] to perform non-local interactions. Messages destined for remote port are directed from the kernel to the internode port and onto the ipcout process which prefixes additional control information to the message before passing it on to the socket driver for transmission. Incoming messages are passed by the socket driver to special buffer processes that strip the control prefix information from the message and pass the message directly to the recipient process. If the incoming message requires a reply the buffer process blocks awaiting the rely from the recipient message. Once received this is passed to the socket driver for transmission back to the sender. The structure of the comms system is:

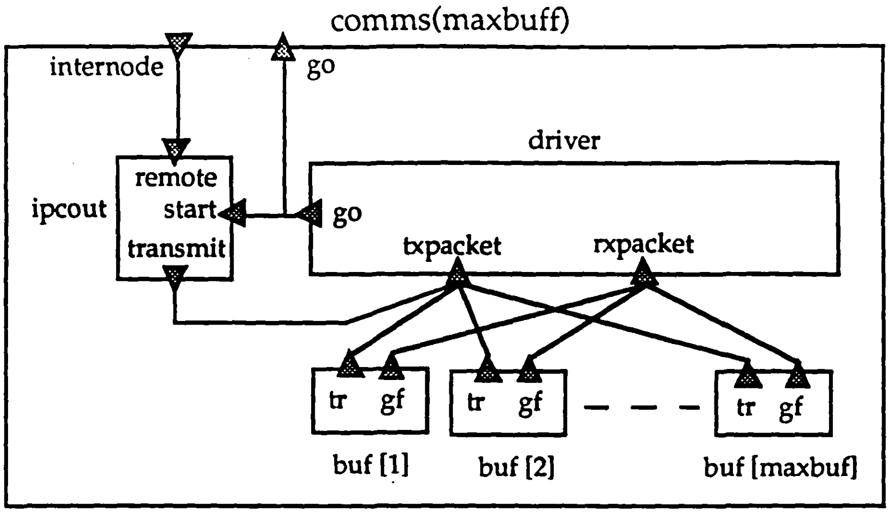


Fig 4.7 Comms Group Module

The go port is used to inform processes that the socket driver is ready to transmit data. The configuration description for the socket driver is shown below where buf=buffer, gf=getframe, tr=transmit

```
group module comms (maxbuf:integer=8);

use       ipcin;        ipcout;        socdriver;

create    ipcout;
          driver:socdriver;

create    forall k:[1..maxbuf]
          buffer[k]:ipcin;

link      forall k:[1..maxbuf]
          buffer[k].getframe    to driver.rxpacket;
          buffer[k].transmit    to driver.txpacket;

link      ipcout.transmit       to driver.txpacket;

-- external interface

use       ipc: bufferT;
entryport internode:bufferT reply signaltype;
exitport  go:signaltype;

link      internode             to ipcout.remote;
          driver.go             to ipcout.start,go;
end.
```

The number of buffer processes is dimensioned according to the parameter passed. If no parameter is passed, 8 buffer processes are created.

4.4. BATCHER'S BITONIC SORTER

The final example is an implementation of Batcher's bitonic sorting algorithm [Batcher68] and illustrates the use of recursive configurations. The algorithm is specifically designed for parallel execution, when it can sort N elements in time $O(\log_2 N)^2$.

The algorithm uses a sophisticated network of primitive comparator elements that take 2 numbers as input and output the minimum and the maximum of the two numbers as outputs. In Conic this can be represented by a task:

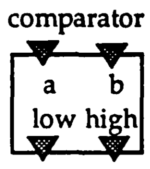


Fig 4.8 Comparator Process

Batcher's sorting network takes an unsorted sequence of numbers, transforms this into a bitonic sequence, and then transforms the bitonic sequence into a sorted sequence.

UNSORTED (S) -> BITONIC (S) -> SORTED (S)

A sequence is bitonic if it consists of two subsequences, one ascending and the other descending, or the sequence can be cyclically shifted into two such subsequences. The ascending and the descending sequences can be generated from the original sequence by subdividing it into two, sorting each subsequence separately, and reversing the elements in the second sorted subsequence, ie:

$$\begin{tikzcd}
 & \text{SUBSEQ-1} \arrow[r] & \text{SORT} \arrow[rrd] & & \\
\text{SEQ} \arrow[ru] \arrow[rd] & & & & \text{BITONICSEQ} \\
 & \text{SUBSEQ-2} \arrow[r] & \text{SORT} \arrow[r] & \text{REVERSE} \arrow[ru] &
\end{tikzcd}$$

Fig 4.9 Bitonic Sorting Algorithm

Thus for sorting 8 elements we would have the following sorting network:

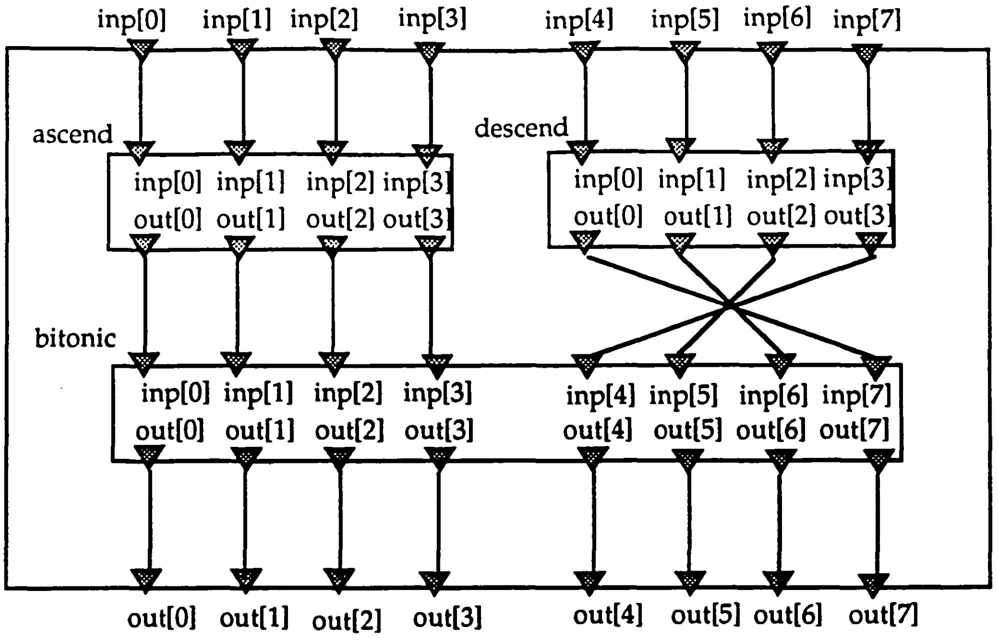

inp[0] inp[1] inp[2] inp[3] inp[4] inp[5] inp[6] inp[7]
ascend
descend
inp[0] inp[1] inp[2] inp[3]
out[0] out[1] out[2] out[3]
inp[0] inp[1] inp[2] inp[3]
out[0] out[1] out[2] out[3]
bitonic
inp[0] inp[1] inp[2] inp[3] inp[4] inp[5] inp[6] inp[7]
out[0] out[1] out[2] out[3] out[4] out[5] out[6] out[7]
out[0] out[1] out[2] out[3] out[4] out[5] out[6] out[7]

Fig 4.10 Sort Group Module for N=8

Each subsequence (ascend and descend) is sorted with its own recursive sort subconfiguration. The configuration language description for sort is:

```
group module sort(n:integer);

entryport  input[0..n-1]:integer;
exitport   output[0..n-1]:integer;

use        bitonic;
create     bitonic(n);

when n>2 create
           ascend:sort (n=n div 2);
           descend:sort (n= n div 2);

when n>2 link forall k:[0..(n div 2) -1]
           input[k]             to ascend.input[k];     -- subsequence 1
           input[(n div 2) +k]  to descend.input[k];    -- subsequence 2

           ascend.output[k]     to bitonic.input[k];    -- sorted subsequence 1
           descend.output[(n div 2)-1-k]                -- reverse of sorted subseq 2
                                to bitonic.input[(n div 2)+k];

-- base case
when n=2 link forall k:[0..1]
           input[k]             to bitonic.input[k];

-- always
link       forall k:[0..n-1]
           bitonic.output[k]    to output[k];
end.
```

This configuration illustrates (i) the declaration and use of arrays of groups ports, (ii) the use of multiple recursive instantiation, (iii) the use of guarded declarations to stop infinite recursion. For the base case, n=2 we would have the following configuration structure for sort:

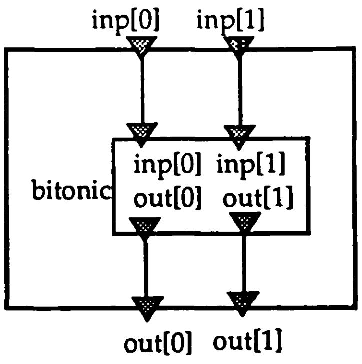


Fig 4.11 Sort Group Module for Base Case

The bitonic part of the network structure relies on the following recursive rule due to Batcher:

A network for sorting a bitonic sequence of 2n numbers a_1 to a_{2n}, can be constructed from n comparison elements and two bitonic sorters for n numbers. The comparison elements must form the two sequences (1) min (a_1,a_{n+1}), $\min(a_2,a_{n+2})$ ... $\min(a_n,a_{2n})$ and (2) max (a_1,a_{n+1}), $\max(a_2,a_{n+2})$ ... $\max(a_n,a_{2n})$. Batcher shows that each of these two sequences is also bitonic, therefore they can each be recursively sorted by a bitonic n sorters. Finally since no number of (1) is greater than any number of (2) the output of one bitonic sorter is the lower half of the sort, and the output from the other is the upper half.

The network for an 8 element bitonic sequence is:

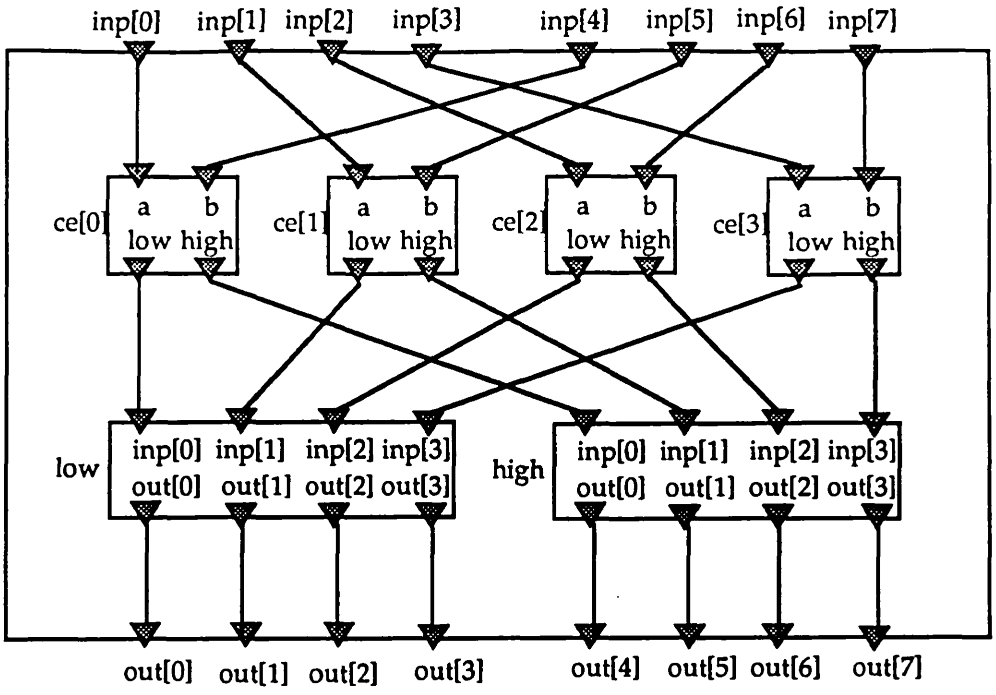

inp[0] inp[1] inp[2] inp[3] inp[4] inp[5] inp[6] inp[7]
ce[0] a b low high
ce[1] a b low high
ce[2] a b low high
ce[3] a b low high
low inp[0] inp[1] inp[2] inp[3] out[0] out[1] out[2] out[3]
high inp[0] inp[1] inp[2] inp[3] out[0] out[1] out[2] out[3]
out[0] out[1] out[2] out[3] out[4] out[5] out[6] out[7]

Fig 4.12 Bitonic Group Module for N=8

The configuration language description for bitonic networks is:

```
group module bitonic(n:integer);

entryport  input[0..n-1]:integer;
exitport   output[0..n-1]:integer;

use        comparator;

create     forall k:[0..(n div 2)-1]
           ce[k]:comparator;

link       forall k:[0..(n div 2)-1]
           input[k]              to ce[k].a;
           input[(n div 2)+k]    to ce[k].b;

when n>2 create
           low:bitonic(n= n div 2);
           high:bitonic(n= n div 2);

when n>2 link forall k:[0..(n div 2)-1]
           ce[k].low             to low.input[k];
           ce[k].high            to high.input[k];
           low.output[k]         to output[k];
           high.output[k]        to output[(n div 2)+k];

when n=2 link
           ce[0].low             to output[0];
           ce[0].high            to output[1];
end.
```

For the base case, n=2 we would have a configuration consisting of a single comparator:

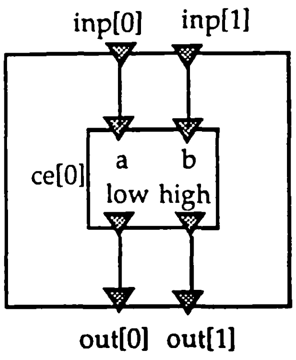


Fig 4.13 Bitonic Group Module for Base Case

In order to complete the sorted, we need an interface module that reads in $n=2^p$ numbers passes them to the sorting network, and outputs the result.

```
group module batcher (n:integer=8);

use      executive;          sort;          interface;

create   executive;
         sort(n);
         interface (n);

link     forall k:[0..n-1]
         interface.out[k]    to sort.input[k];
         sort.output[k]      to interface.inp[k];
end.
```

Although this example illustrates the expressive power of the configuration language, the algorithm itself is quite unsuited to todays technology. For example a bitonic sorter for 2^p numbers would require $(p^2+p)2^{p-2}$ comparators. A sequence of length 256 would require 4608 comparators, while a sequence of length 1024 would require 28,160 comparators. Nevertheless, this and similar algorithms can be readily expressed within our language. This example is also interesting in that the structure of the program is of far more interest and complexity than the algorithmic parts.

4.5. CHAPTER SUMMARY

This chapter has demonstrated the power and applicability of the configuration language described in Chapter three with a series of graduated examples that illustrated many features of the language.

The first example, a simple flow control pipeline, introduced group module definition, context definition, component instantiation, component bindings, parameter declarations and parameter passing.

The second example, the structural part of a solution of the dining philosophers problem, demonstrated the use of replicated instantiation and binding, as well as the use of parameters to control the dimensionality of replicated forall declarations and in the declaration of a ring structure.

The third example, a run-time executive for Conic, introduced the use of hierarchic structuring: groups with subgroups, groups ports and hierarchic binding. This example also highlights an implementation-defined rule for performing default bindings of standard ports based on the bindings defined for the standard ports of the kernel instance within the executive.

The final example, an implementation of Batcher's bitonic sorter demonstrates the use of recursive structuring: recursive instantiation, guarded instantiation and guarded binding. The example also uses replicated ports and replicated port bindings.

All the examples described have been successfully run within a distributed environment. The next chapter introduces the implementation model used to execute distributed programs constructed with our configuration language. Chapters 6 details the techniques used for the compilation of programs, while Chapter 7 continues with the techniques used for the execution of programs.

Chapter Five Implementation Model

This chapter introduces an implementation model for the Conic configuration language described and demonstrated in the previous two chapters. The model is designed to act as a framework for constructing efficient implementations of configuration-based distributed programs. The chapter gives an overview of the distributed environment in which the language is used, and discusses the main design objectives for our implementation. The overall structure and strategies of the model are then introduced.

5.1. INTRODUCTION

The use of a powerful configuration language for structuring distributed programs raises many difficult and interesting implementation issues, for example, can configuration specifications be compiled, if so how are compiled configuration specifications elaborated, how should group modules be distributed, at what levels of specification should dynamic reconfiguration be supported, how can data to be sent between heterogeneous processors, how can the current configuration structure of a program be queried?

The model has been influenced by a number of factors, in particular, by the Conic distributed programming environment in which it works. This environment is particularly interesting for its emphasis on supporting heterogeneity. Although our implementation model was developed for this environment it is sufficiently general to be adaptable to other environments also.

Distributed programs in the Conic environment are developed for a network of computers. Logically this network can be viewed as follows:

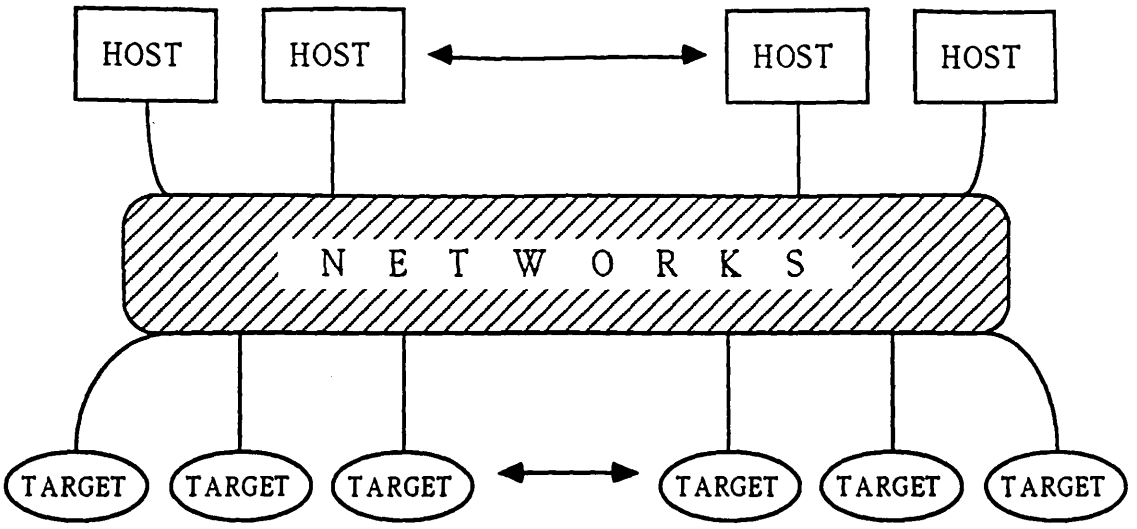


Fig 5.1 Host/Target Environment

Conic programs can run on both hosts and targets. Hosts are used for program compilation, execution, and debugging. Hosts also run the Conic dynamic configuration management software, and provide file and terminal I/O services for targets. Targets are typically used for executing components that control particular hardware, or components that require a predictable response. Targets have also been used for their raw performance capability, since they do not have a host operating system overhead.

All the hosts currently run some variant of the Unix operating system [Bach86, Leffler89]. This is both an advantage and a disadvantage. The advantages stem from being able to exploit underlying host mechanisms in the implementation model. For example, all our hosts provide sockets, a network-transparent IPC mechanism. Obviously this may not be possible in non Unix environment, or mixed operating system environments, where bridging software may be needed. The disadvantages stem from the lack of mechanisms for concurrent and distributed program execution, for example, many systems still do not support light-weight processes, dynamic loading of object code, remote execution or process migration. These mechanisms are hard to provide without extensive modifications of the underlying Unix kernel. A target-only execution environment is obviously better suited to developing such mechanisms. We have attempted to address some of these issues directly, for example, with the provision of a nodes, which provide fast light-weight process execution, and virtual targets which enable distributable parts to be created on remote computers without the overheads of logging onto the remote computers.

The network currently consists of 25 SUN 3 hosts running SunOS 3/4 Unix, 3 DEC VAX hosts running BSD 4.3 Unix, 11 HP 9000-300 hosts running System 5 Unix, 8 Motorola MC680x0 targets controlling a variety of real-time equipment, and 1 DEC LSI-11 target. These are interconnected via multiple Ethernets. Users can, and do, develop configuration-based distributed programs that run across a heterogeneous selection of these computers.

5.2. IMPLEMENTATION OBJECTIVES

The implementations objectives for the model are summarised below:

Efficient implementation.

Configuration-based distributed programs should be as efficient as other styles of distributed program.

Portable implementation.

The implementation model should produce portable implementations. Components should be runnable on a variety of hosts and target processors.

Support distributed development.

The implementation model should support distributed development of programs. It should be possible to develop parts of distributed program in isolation, and then configure them into a running program.

Tool Support.

The implementation model should utilise existing tools and resources where appropriate. The model should permit the integration of new tools in simple and coherent manner.

Since an efficient implementation is also more likely to be used than an inefficient one, efficient techniques are also needed to convincingly demonstrate the scalability of the language and implementation model. The implementation model aims to minimise file costs, memory usage, execution time, and network communication costs.

Portability is an important implementation objective in such environment. A high degree of portability has been achieved by adopting a high-level language approach to implementation. Rather than the traditional approach of generating low-level code, the configuration language (CL) compiler generates "human-readable" procedures In addition all the tools, and run-time support components have themselves been configured with the Conic CL and programmed with the

Conic programming language (PL).

Our implementation model supports distributed development and distributed execution of programs. It allows components to be developed entirely independently by different people on heterogeneous hosts, and then be safely configured into a running system. Target components can be cross-compiled, downloaded into a bare machine, and linked into a running system. The range of configuration options available to users is the same for components running on targets as for components running on hosts.

Existing mechanisms and tools have been adopted whenever this was cost-beneficial to the implementation effort. This is important and gives the implementation model a looser, and more flexible structure. The CL compiler, for example uses the standard Unix link-loaders. A utility that generates Makefile descriptions[7] from configuration descriptions is also provided.

5.3. NODES

The key issue addressed by the model is at what level components are to be run, distributed, and reconfigured. This is itself dependent on the nature of the execution environment. Ideally one should be able to assign each component to its own processor. This is rarely possible in practice, since the number of components in a program normally outnumber the number of available processors. Next, one could consider automatically partitioning the components into sets according to the number of processors, and let each processor, multi-program the components in its assigned set. This approach is better suited to homogeneous shared multi-processor environments, where there is no advantage in having control of where components are placed. Where the user has special knowledge of the mapping constraints of his program and environment, a technique is needed to let users control the placement of components.

One approach is to allow users to specify locations in the create clause of the configuration language. Each create clause could then optionally specify a location for the created component instance, for example in

create *instance:type (parameters)* [**at** *location*]

location could specify either another component instance with which the created instance was to be co-located or an actual machine designator (e.g. the machine

7 Make is a tool for controlling the initiation of a sequence of Unix commands, based on the dependency relations and modification dates of files.

name or machine internet address) at which the created instance is to be created.

```
create A: At at 129.21.12.32
create B: Bt at A
```

If the location were omitted, the created instance could be located at the same location as the encapsulating group instance, i.e the location would be deferred until an instance of the group type is created. In this way each component in the program can be labelled with its eventual location.

This is quite an attractive approach for some programs, but was rejected in favour of simpler less predetermined approach that deferred such decisions until run-time.

In Dicon [Lee86], processors are assigned exclusively to particular programs unless the processor is specified as shared in the configuration specification. It is also possible to state that two components are to be co-located.

Another possibility would be to define a separate mapping language for specifying the location of components [DeRemer76]. The disadvantage with this approach is that it requires the mapping language to be as expressive as the configuration language. An important consideration in mapping components in heterogeneous networked environments, is the relatively large differences in communication costs between components running on the same machine and components running on different machines.

A second important consideration in deciding on the unit of distribution is the relatively poor implementations of processes provided by Unix operating systems, which often limit the number of processes allowed, and impose relatively high scheduling and interprocess communication costs on the processes they support. These constrain the degree of concurrency allowed, and have been reduced by providing light-weight processes within a Unix process, e.g. Unix processes with over 100 Conic tasks have no noticeable affect on the performance of Unix .

For simplicity, and because of the considerations discussed above, the implementation model defines a distributed program to consist of a flat network of interconnected nodes, where nodes are group modules whose sub-instances are loaded within a single address space when the node is created. Nodes thus act as the unit of distribution within the model, with the actual distribution of nodes left to the responsibility of users.

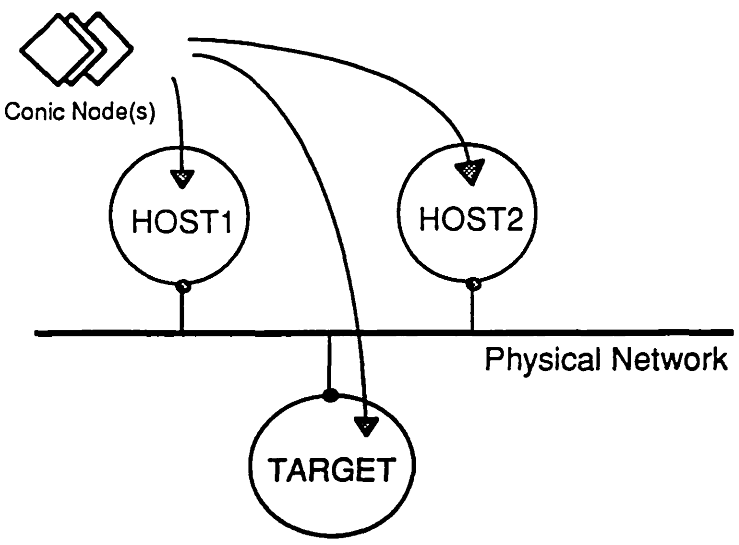


Fig 5.2 Node Distribution

The number of components (groups and tasks) allowed within a node is limited only by the memory limitations imposed by the running computer. Each node includes a special executive sub-component which is responsible for all aspects of node management, including intra-node component interaction and scheduling. On Unix machines nodes are run as Unix processes.

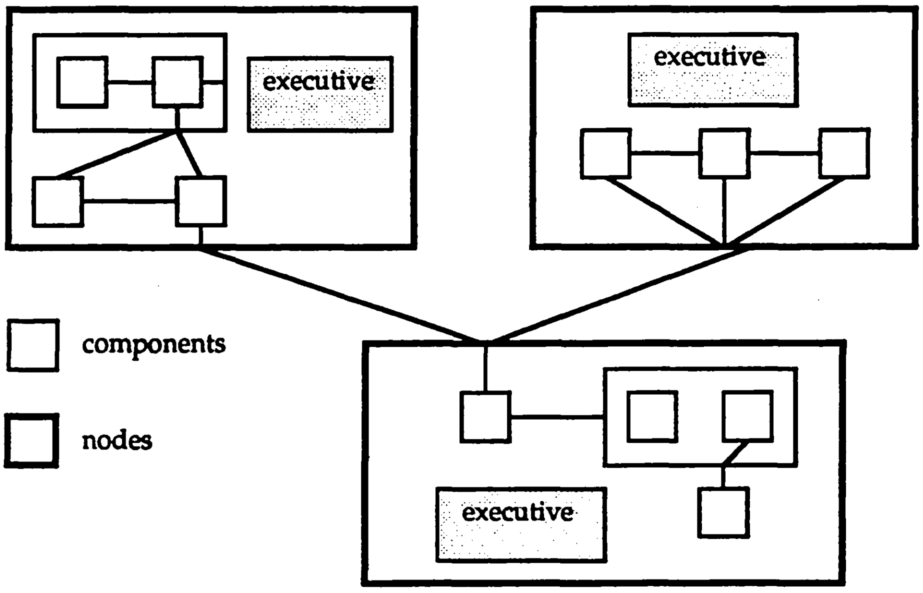


Fig 5.3 Distributed Programs with Nodes

Nodes are also the unit of reconfiguration supported by the model. Once distributed and started, the node executive periodically reports the status of the node to a global name server. This maintains a register of all reporting nodes and their location. Nodes can then be configured by interactive tools called configuration managers, that query the server for the names and locations of nodes within an program. Two configuration managers are provided, iman that runs on Unix hosts and has a textual command interface, and ConicDraw that runs on Apple Macintosh® 's and has a graphical iconic interface. ConicDraw interacts with the rest of Conic system through gman to which it linked over a serial link. Gman is a specially modified version of iman that runs on Unix hosts and which performs the actual configuration queries and commands on behalf ConicDraw. The configuration management system is outlined below:

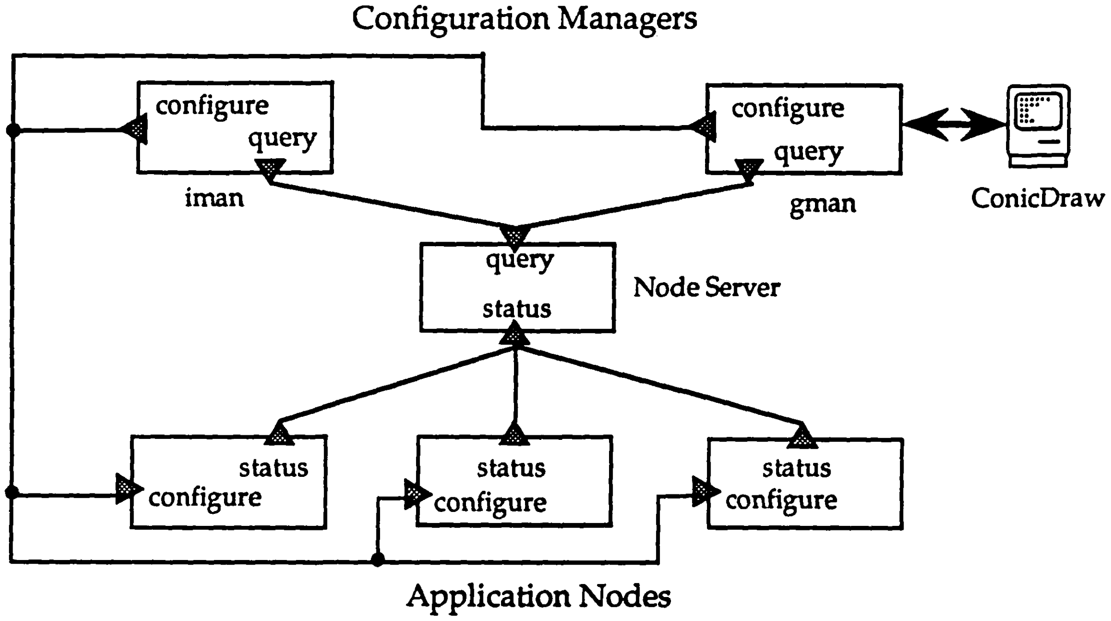


Fig 5.4 Structure of Configuration Management System

Once the location of a node is known it can then be directly queried, instructed to bind or unbind ports, stopped, restarted or removed from a configuration manager.

5.4. DEVELOPMENT CYCLE

The development cycle of a distributed program within the model consists of a node compilation phase, in which a runnable node is produced from a set of compiled sub-components and a run-time phase, in which the nodes produced from the compilation phase are distributed, elaborated, and configured into a running program:

® Apple is a registered trademark of Apple Computer Inc.

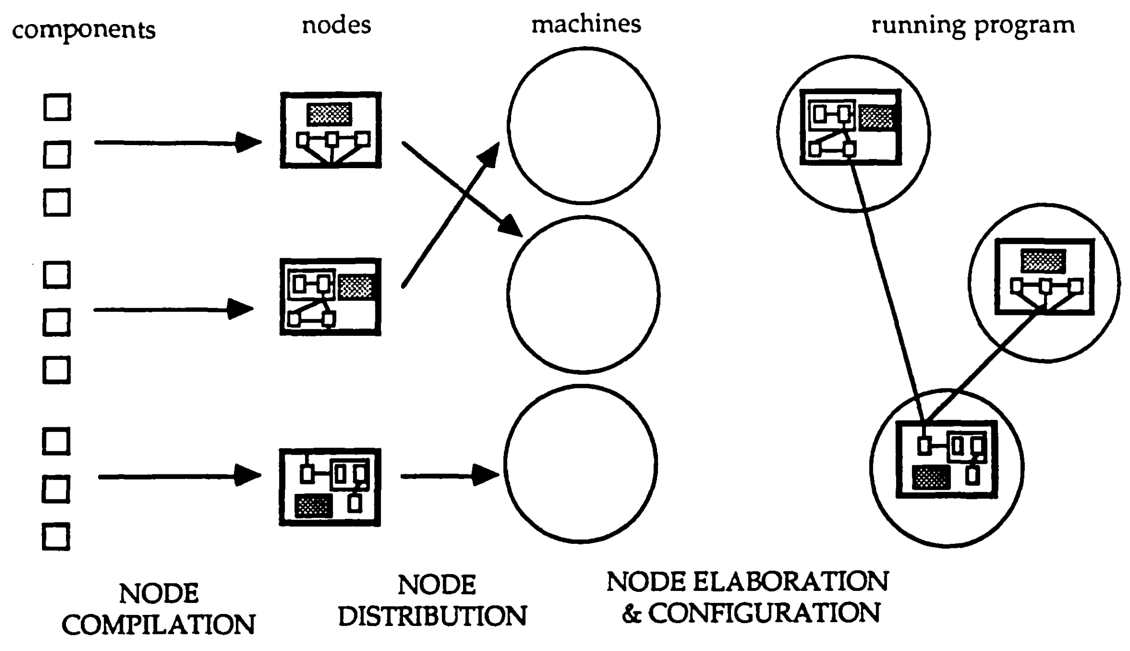


Fig 5.5 Node Development

Once started, running programs can be re-configured by interacting with a configuration manager. New nodes can be compiled and configured into the program while existing nodes can be relinked or removed from the program:

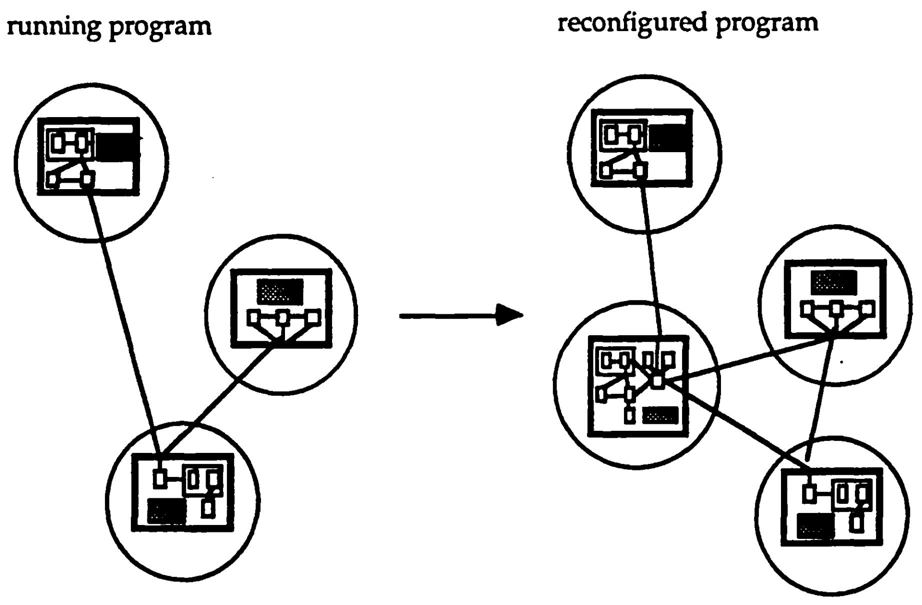


Fig 5.6 Node Reconfiguration

5.5. AN EXAMPLE

The components in the dining philosophers example given in Section 4.2, can be grouped into diner nodes and a table node:

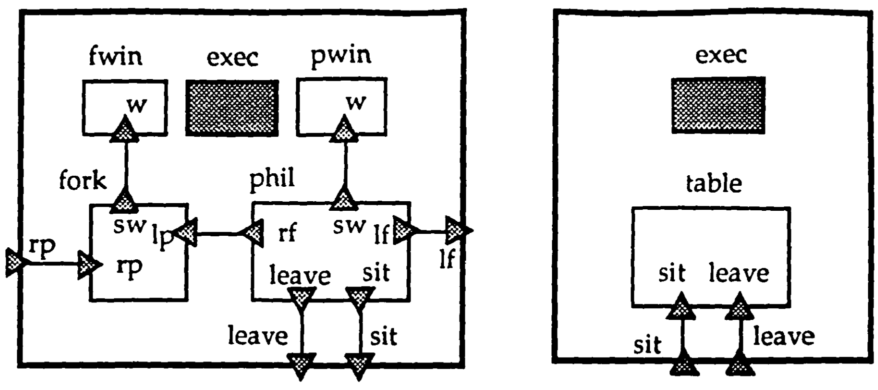


Fig 5.7 Diner and Table Nodes

and the nodes configured as:

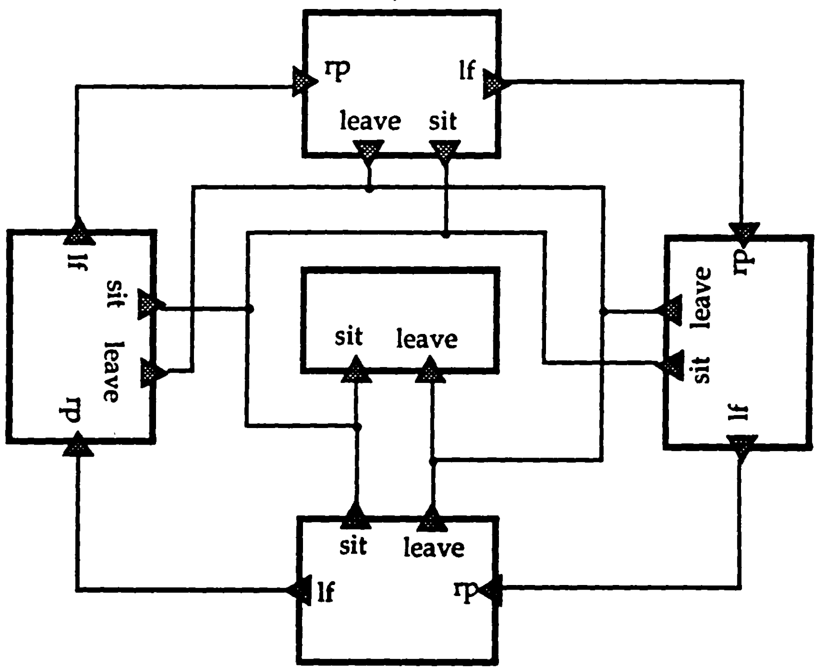


Fig 5.8 Dining Philosophers System as Configured Nodes

Further details on how to configure this example are given in Section 7.5.

5.6. CHAPTER SUMMARY

This chapter has introduced an overall model for the implementation of our configuration language within a distributed environment. The model makes a number of simplifying assumptions in order to allow efficient implementations to be developed. A unit of distribution and reconfiguration called the node is defined. Nodes are groups whose sub-components will share a single address space. On Unix hosts, nodes map onto Unix processes. There are no restriction on the number of sub-components allowed within a node, or the number of nodes within a program, except those due to lack of memory or operating system resources. The mapping of nodes to processors is left to users, although tools are also provided.

For presentation the tools and techniques used in the model have been loosely grouped into those associated with language compilation and those associated with run-time execution and support. Chapter 6 describes the techniques for language compilation, and Chapter 7 continues by describing the techniques used for run-time execution and support.

Chapter Six Compilation Techniques

6.1. INTRODUCTION

This chapter presents techniques to support the separate compilation of group modules and definition modules that are efficient and ensure early detection of the use of inconsistent types and components. In order to cater for distributed programs built from large numbers of separately compilable units, the techniques employ a new symbol file design for efficiently representing the interfaces of separately compiled units. Our symbol files are also used to automatically track object files in the host file system, alleviating the programmer from explicitly having to specify which objects files will be required to link-load a node. A new technique for performing type extension checks in constant time rather than linear time is also presented. Our techniques are fast, simple and scalable.

Unlike other systems, which elaborate (instantiate and bind) program structures at link-load time, we adopt a more dynamic approach and elaborate the structures within a node at execution time. This is done by generating object code to elaborate group modules. This results in a system that is both very fast and very flexible, particularly when parameterised structures are required. Further the generated code can be called dynamically to create new structures while the node is running.

6.2. NODE COMPILATION

To build a runnable node, each sub-component of the node is separately compiled. Groups are compiled by the configuration language (CL) compiler, which produces code to elaborate instances of the group, processes are compiled by the programming language (PL) compiler, and definition modules are compiled by a definition module compiler[8].

8 In fact definition modules are currently compiled by the PL compiler.

Once all sub-components have been compiled, the node group is itself compiled by the CL compiler, and all object files link-edited by the host or target link-editor into a runnable executable.

A number of files are produced when a component is successfully compiled: a *symbol file* for all processes, groups, and definition modules, an *object file* for all processes and groups, and *linker file* for all node groups. The transformations used to produce nodes is outlined below:

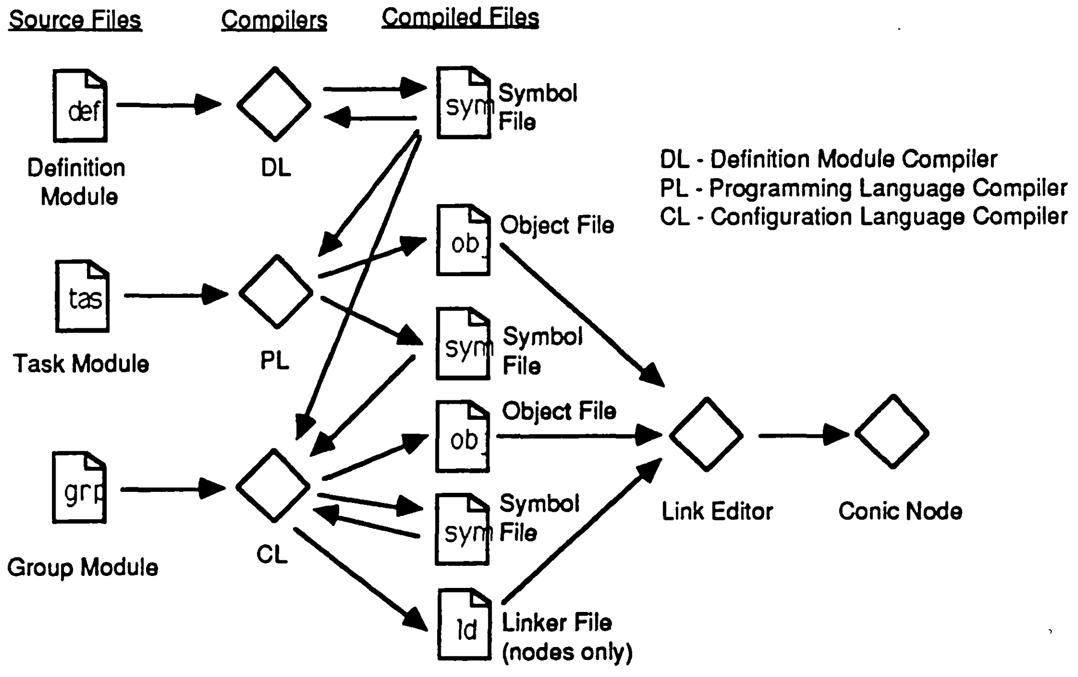


Fig 6.1 Node Compilation System

Symbol files hold a special compiled form of the component, used when the component is referenced in further components. Symbol files also keep track of the object files that will be needed in final link loading. Object files hold the machine code version of a component. Linker files hold the names of all files that the link-loader will require to create a node.

When it is not possible to modify the PL compiler to directly read and write symbol files, an alternative technique would be to use the CL compiler to describe the process, and to add a pragma in the description to indicate the name of the object files that implement the process as is done in Dicon [Lee86]. For example:

```
group module fortranProcess;       {Configuration Description for a PL component}
objects "fortran.o";               {Object file pragma}
use     <PL Components>            {Each PL component is described by a similar}
                                   {description}
```

```
            <Port Declarations>

        -- No BindingAllowed
        end.
```

The Conic PL is currently used to describe the interface of components written in other programming languages, for example, Fortran and C.

The host Unix link-editor, ld, is used to generate the final runnable node. Cross-compilation to a machine that is different from the host, requires a machine-specific linker.

6.3. SYMBOL FILES

Symbol files hold a special compiled form of a component, read by compilers and debugging tools whenever a description of a component or of items within a component is required, for example, whenever a 'use' declaration is encountered. Symbol files are distinguished from object files, in that they do not normally include executable code, only information for use during compilation. A good implementation of symbol files is essential if the system is to be used for the practical construction of very large programs. In line with Lampson's advice in [Lampson83], our implementation aims to fast and simple. It also aims to be scalable .

Symbol files are used to ensure that strong type checking is maintained across separate compilations, that inconsistent use of components is detected and reported as earlier as possible in compilation, and that visibility rules are preserved.

In contrast to source code based symbol files our symbol files abstract out the essential details of a component, by recording the internal representations of items built by compilers during compilation in binary form. They are thus faster to process and more compact than systems that use modified source files [Foster86], since they do not require each compiler to have a parser for the type description language. In a "sense", they can be considered as a medium for holding completely abstract language independent descriptions of objects such as components and types.

Our symbol files also attempt to track the location of associated object files in the host file system. This is a very convenient facility and alleviates the programmer from explicitly having to specify which objects files will be required to link-load a node.

Our symbol files are also used for implementing a new technique for performing sub-type compatibility checks. The technique trades symbol file space and some static type-hierarchy evaluation for fast constant-time run-time evaluation.

Our symbol files are *complete*. Reading one symbol file does cannot cause further symbol files to be read. Our symbol files are also *minimal*. Only information that is directly relevant to exported items is recorded. In contrast to [Sweet85] our symbol files occupy little space.

Since symbol files are written once but often read many times, our symbol files are optimised for reading. They need only be read once during a compilation, and are organised for one-pass sequential reading without backtracking.

In [Robbins84] a single random access, global symbol table database is built and updated for each program. This solution has the advantage of opening a single symbol file for user-written component, but becomes costly when pre-existing symbol files have to be merged into the program's symbol file. The central database idea is also advocated in [Rudmik82] which also stores intermediate code in the database.

6.3.1. Symbol File Organisation

Symbol files are organised into the following sections:

Symbol File

Magic Number Section
Directory Table Section
Component Section
Parameters Section
Identifier Section
Type Extensions Section
Debugging Section

The key sections are the component section which holds information on the component and all components used by the component, and the identifier section which holds information on all items exported from the component, and on any auxiliary items that are needed to complete description.

6.3.2. Symbol File Syntax

The format of symbol files is given in a boxed form of BNF. Terminal symbols are written in UPPERCASE. Non-terminals are written in lowercase with the first letter of each word in upper case. Non-terminals may be prefixed by a label and colon (:). Labels are used purely for exposition. A * suffix denotes zero or more repetitions. A + suffix denotes one or more repetitions. Productions are shown boxed, with the rule specified in the box and the non-terminal above the box.

6.3.3. Magic Number Section

Each symbol file begins with a two byte number, called the magic number.

Magic Number Section
Magic Number : Integer

The magic number is used to prevent other tools, particular text processing tools, such as editors from attempting to read the file. On Unix, such tools, often check that the first byte of a file is an acceptable Ascii code before reading the remainder of the file. For compilers, the magic number provides a useful check that the file to be read is actually the kind of file expected, and not some other kind of file. The following magic numbers are currently employed:

Definition module symbol files:	21923
Task Module symbol files:	21924
Group symbol files:	21925

Changing the magic numbers accepted and written effectively invalidates existing classes of symbol files. This may be useful where a major revision of the programming system has been made, and the implementors require all existing components to be re-compiled.

6.3.4. Directory Table Section

A table of file directory names follows the magic number.

Directory Table Section
Length : Integer
Directory Name: String *

This table is used as a hint in tracking the location of object files for subsequent link-loading into an executable node. The table consists of an integer length field followed by that number of strings, each string holding the pathname of a Unix directory, for example:

```
3
/usr/lib/conic/module
/users/nd/example/ward
/usr/lib/conic/sort
```

Directory names are implicitly numbered from 1 onwards, and the numbers used as references in the component descriptions of the component section.

Duplicate directory names are not permitted to occur within a directory table. This rule greatly reduces the size of symbol files when many components occur in the same directory. For example a distributed program having 20 components in one directory, where the length of the directory name is 50 characters, would take up one 50 character string, plus 20 2-byte indices, rather than 20 times

50=1000 characters, a saving of 910 bytes.

6.3.5. Component Section

The component section holds the component description of the compiled component (the first description) along with the component descriptions (the remaining component descriptions) of all components directly and indirectly referenced by the component. Descriptions of dependent components are used to check consistent use of components during compilation, and are also used to track the location of object files automatically. The component section consists of

Component Section
Comp : Component ID + Null

where a Component ID (Identifier Description) is

Component ID
Name : String
Comp Number : Integer
Home : Integer
Time Stamp

The name field identifies the name of the component type. The component number field provides a reference number for item descriptions in the Identifier Section, and type extension relationships in the Type Extensions Section. The home field specifies in which directory the component was compiled, and therefore in which directory its object file may be found. The home field indexes the nth directory specified in the directory table part of the symbol file. Timestamp is a 6-byte tuple that uniquely identifies the component. Timestamps are created when the component is compiled and consist of:

Time Stamp
Epoch : Longint
Unix Process Id : Integer

On Unix, the time in seconds since 1970, is used as the epoch, while the Unix Process-Id of the compiler process is used to differentiate between two compilations started during the same epoch second.

In addition to the fields above, component type descriptions hold other attributes such as whether a component is executable or not, and estimates of the minimum memory requirements of a component.

6.3.5.1. Consistency Checking Algorithm

Timestamps are used to check that interdependent components are consistently used. The algorithm used to check consistency follows.

Each compiler maintains a table of all component descriptions read from imported symbol files. Each time a component type description is read, the component type table is searched for a component of the same name. If no entry is found, then the read component is added to the end of the table. If an entry is found, then the timestamp of the stored component is compared to the timestamp of the read component. If the timestamps are unequal then this implies inconsistent versions of a component and gives rise to a compilation error. An array which maps symbol file component numbers to compiler components is used during symbol file read.

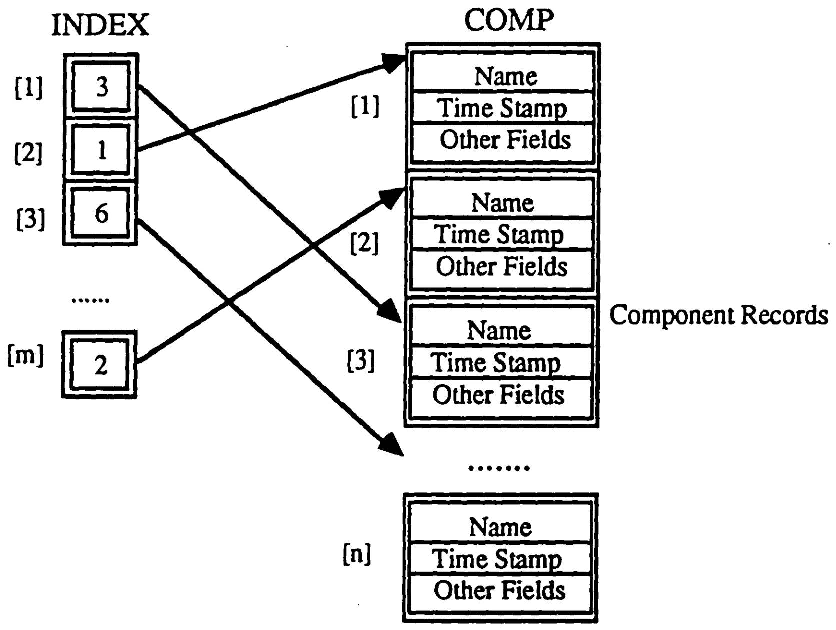


Fig 6.2 Organisation of Component Records

COMP	Table of known components known to compiler. COMP [1] holds the information for the component being compiled.
INDEX	An array of indexes to component records in COMP. INDEX is used to map symbol file component numbers compiler to component numbers. A new INDEX map is need for each symbol file read.

```
read Nth Component ID into C
if C.name in COMP
then    let E=Entry found in COMP
        if C.timestamp = E.timestamp
        then Let INDEX [N] = Index of E in Comp
        else Dependency Error
        endif
else    let E=New Entry in COMP
        Copy C to E
        let INDEX[N] = Index of E in Comp
endif
```

Checksums can be used as an alternative to timestamps, [Bron85] suggests the use of a component checksum calculated on the exported items of a component instead of timestamps as component signatures and link-time checking instead of compile-time checking. Thus when a component A is link-edited to a component B, a check is made that the code of B bears the same checksum as was valid for B when B was used during the compilation of A. Accidental checksum equivalences can be reduced by the use of longer checksum values. Checksums are quite attractive as a means of determining whether a component may have changed its interface or whether two components have the same interface.

6.3.6. Parameters Section

The parameters section specifies the formal parameters of the first component in the symbol file. When read these parameters are chained together and the first parameter linked into the component description by the compiler.

Parameters Section
Parameter Identifier Null

The format and handling of parameter identifiers is identical to other identifiers described in the Identifier Section below.

6.3.7. Identifier Section

The identifier section of a symbol file is used to record the descriptions consists of named port identifiers, user-defined type identifiers, constant identifiers, record fields and parameter identifiers.

Identifier Section
Identifier * Null

where

Identifier
(Port Identifier | Type Identifier | Constant Identifier | Field Identifier | Parameter Identifier | Nil Identifier)

Identifier descriptions (except for the Nil Identifier) consist of a set of common fields followed by some identifier specific fields.

Common Id Part
Id Name : String Owner : Integer Id Type : Type Structure Next : Identifier

The common fields specify the name of the identifier, a reference to its owning component, a detailed description of the type of the identifier, and a next field which is used to chain a sequence of identifiers together, such as the enumerated constants of an enumerated type, the fields within a record and the parameters of a component. Within the CL compiler each identifier description is translated into a pointer to an identifier record. The Nil Identifier is used to represent and generate a nil pointer.

Nil Identifier
Id Kind : Null

Ports descriptions hold additional information on whether the port is an entryport or an exitport, and a number that indicates the declaration position of the port within the component, i.e the nth declared port.

Port Identifier
ID Kind: PORT
Common Id Part
Port Kind: (EP | XP)
Port Number : Integer

Type descriptions hold no additional information.

Type Identifier
Id Kind : TYPE
Common Id Part

Constant description hold the value of the constant, for simplicity this is held in either integer form or string form. Real constants for example, are held in string form. The type of the constant can be deduced from examination of the Id Type field in the Common Id part of the description.

Constant Identifier
Id Kind : CONSTANT
Common Id Part
Value : (Integer | String)

Field descriptions hold additional information on the offset of the field within the record.

Field Identifier
Id Kind : FIELD
Common Id Part
Offset : Integer

Parameter descriptions hold additional information on the position of the parameter in the parameters declaration.

Parameter Identifier
Id Kind : PARAMETER
Common Id Part
Parameter Position : Integer

The missing productions are defined in Appendix II.

6.3.7.1. Reading Identifier Descriptions

Within the compiler, identifier descriptions (except field names) are attached to their defining components in alphabetically ordered binary trees of identifier records:

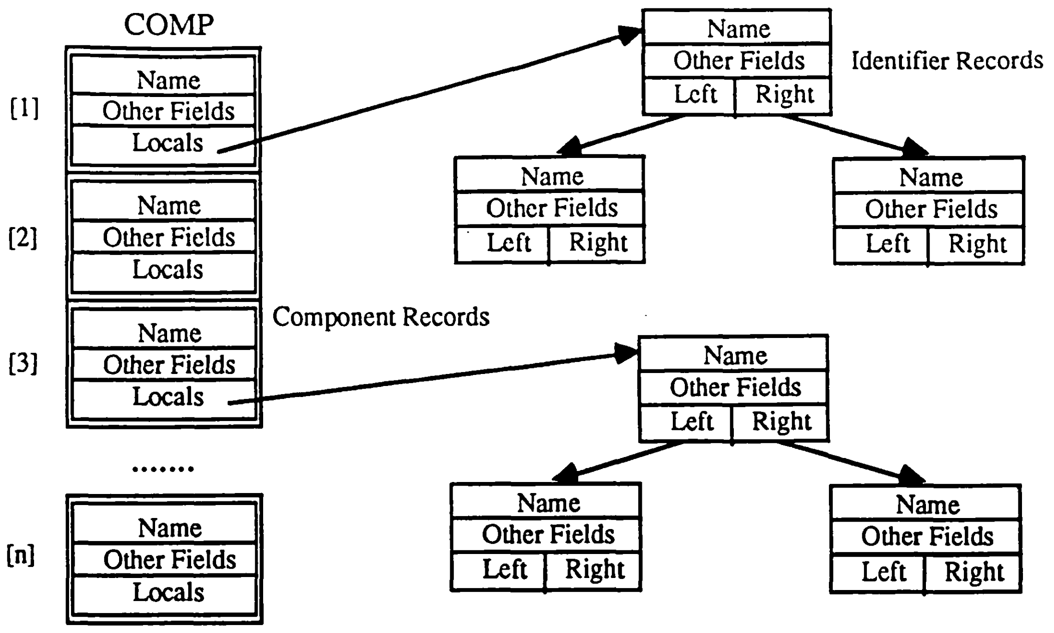


Fig 6.3 Organisation of Identifier Records

The procedure for reading identifier descriptions is thus

```
if next Identifier.Kind=NULL then
        return nil
otherwise
read next Identifier.(Id Kind, Name, Owner) Into I
let I.owner = INDEX [I.owner] {Remap component number}
let Tree = COMP [I.owner].locals

if      I.name in Tree
then    skip over rest of description (Id Type, Next, & identifier specific information)
        return pointer to entry in Tree
else    Insert I into Tree
        read rest of description into I.(Id Type, Next & identifier specific information)
        return pointer to I
endif
```

The procedure for skipping identifiers is identical to the above, except that read descriptions are never inserted into component identifier trees. Identifier chains, for example the Next part of an identifier description, are read by recursively calling the read procedure above.

6.3.7.2. Identifier Visibility

Identifiers are made visible in the scope of the importing component, by inserting a copy of the identifier record into the identifier tree for the importing component namely COMP[1].locals. For enumerated types, all the enumerated constants of the type need to copied.

6.3.7.3. Writing Identifier Descriptions

The procedure for writing an identifier is

```
if Identifier is nil
then    write NULL
else    write Identifier.(kind,name,owner,type,next)
        write Identifier specific information
endif
```

6.3.7.4. Type Structure Descriptions

All identifiers have a type. Type structure descriptions record type information. Types in Pascal can be highly inter-related, for example types can be included within other types, or recursively used. Types can also be mutually interdependent. In order to handle such degrees of type usage the compiler builds and maintains a network of type structures. For example, the type definitions

```
range = 1..80;
colour = (blue, green, red);
rope = record
          len : range;
          seg : array [range] of colour;
       end;
ptr = ^ rope;
```

are represented by the following identifier/type network:

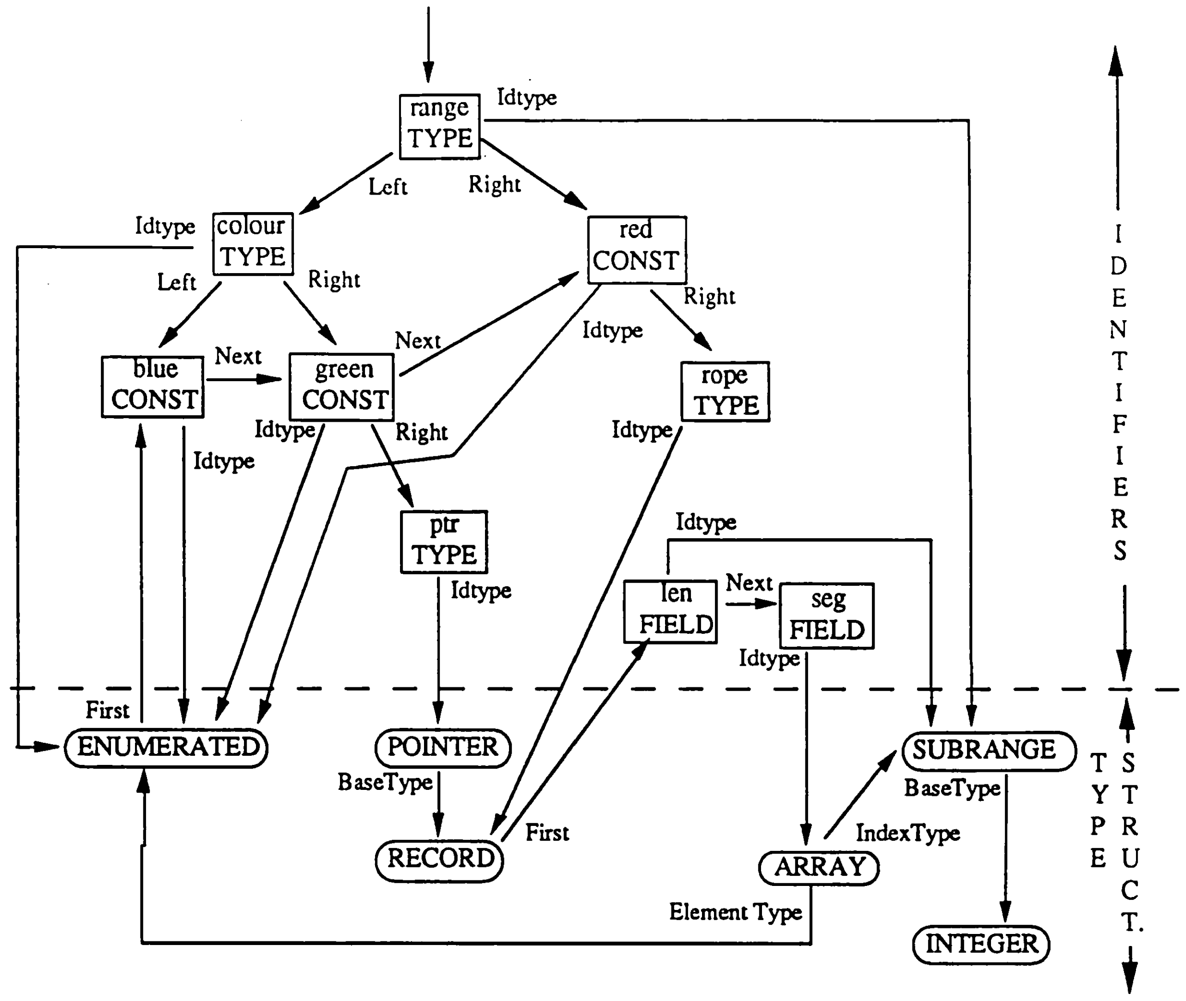


Fig 6.4 Example of an Identifier/Type Network

The type structures in such a network must be traversed and written to a symbol file, whenever an identifier is exported. Note that some type structures, for example, records and enumerated types require identifier descriptions (e.g. of fields and enumerated constants) to be written. Conversely when a symbol file is read the unrolled type structures need to relinked into the compilers network of types. Two identifiers have the same type if they point to the same type structure.

All type structure descriptions are labelled by an integer key. The key is used to indicate whether a type structure is of a standard type (keys 1 to 20), of a user-defined type (keys 21+), or whether the structure is referenced by an type identifier that has not yet been written (key 0).

```
Type Structure
( Standard Type | Embedded Type | Read type |Message Type | Subrange Type |
  Enumerated Type | Array Type | Record Type | Set Type | Pointer Type )
```

The standard type NIL is used to represent a nil pointer.

Standard Type

```
Key : ( NIL | BOOLEAN | CHAR | INT | REAL | BYTE | NATURAL |  LONGINT | STRING
)
```

If a type description has already been written to the symbol file, it is represented solely by the key used in first description of the type.

Read Type
Key : Integer

User defined type structures written for the first time consist of a set of common fields followed by some type structure specific fields. The common fields specify a key (in the range 21+) which is used to identify the particular type structure, the size of the type structure in bytes, and whether the type is packed or not.

Common Type Part
Key : Integer
Byte Size : Integer
Packed : Boolean

Message type structures also record the request type and reply type of the port.

Message Type
Common Type Part
Form : MESSAGE
Request Type : Type Structure
Reply Type : Type Structure

Enumerated type structures also record the first enumerated type constant, which chains the remaining constants.

Enumerated Type
Common Type Part
Form : ENUMERATED
First : Constant Identifier

Subrange type structures also record the low and high values of the subrange, and the base type of the subrange.

Subrange Type
Common Type Part
Form : SUBRANGE
Low Value : Integer
High Value : Integer
Base Type : Type Structure

Array type structures also record the index type and element type of the array.

Array Type
Common Type Part
Form : ARRAY
Index Type : Type Structure
Element Type : Type Structure

Record type structures also record the super type of the record (if an extension) and the first field of the record which chains the remaining fields.

Record Type
Common Type Part
Form : RECORD
Super Type : Type Structure

First : Field Identifier

Set type structures also record the base type of the set.

Set Type
Common Type Part
Form : SET
Base Type : Type Structure

Pointer type structures also record the base type of the set.

Pointer Type
Common Type Part
Form : POINTER
Base Type : Type Structure

A special mechanism for recording embedded type identifiers is needed to enable one-pass writing and reading of symbol files. Embedded type identifier descriptions are signalled by key 0. The key of the new identifiers type follows the identifier description

Embedded Types
Zero Key : Integer
Type Identifier
New Key : Integer

6.3.7.5. Reading Type Structures

Types must maintain their definition across a distributed program, although they may only be visible in some of the components.

The following data structure is needed during each symbol file read:

STRUCT	An array of pointers to read type structures records. STRUCT is used to map keys within the symbol file to type structure records within the compilers type network. STRUCT [1..20] is reserved and used to map the keys of the standard types to standard type structures records.
HighKey	is initialised to 20 at the start of each symbol file read, and updated each time a new user-defined type structure is read.

The procedure for reading a type structure is then

```
read    Key
if      Key=0
then    read Type Identifier I              { Embedded Type }
        read key
        Point STRUCT [Key] at I.(Id Type)
elsif   Key > HighKey                       { New User-defined Type }
then    read rest of structure into a new type structure S
        Point STRUCT [Key] at S
        HighKey := Key
else                                        {Standard Type or Read Type }
endif
return STRUCT[key]
```

The procedure for skipping type structure is identical to the above, except that a dummy type structure S is used, and the skip identifier procedure used for embedded types.

6.3.7.6. Writing Type Structures

When a type structure is written for the first time it is preceded by the next available key, which is saved in the type structure. On the second and subsequent writes, only the saved key is written out. Keys are generated in ascending order starting from 21.

The keys of user-defined type structure are initially zero. They keys of standard type structures are initialised to a value in the range 2 to 20. Key 1 is used to indicate a **nil** pointer.

Key 0 is a special key that allows a type identifier description to be embedded with a type structure description. This is needed to enable one pass traversal of the compiler identifier/type network. It is also necessary to read types in dependency order. If a type A refers to another type B, then the description of B must precede the description of A. This however would require a search of the type dependencies before the type is written. To avoid this, we allow a type structure description to be interrupted mid-stream with declarations of referred identifiers.

The procedure for writing a type structure S is then

```
if      S=nil
then    write 1
elsif   S.key > 0          { Standard Type or previously written User Defined type }
then    write S.key
elsif   a type identifier I points to S and has not yet being written
then    write 0
        write identifier I
        set S.key to next key
        write S.key
else    set S.key to next key
        write S.(key, common part, specific part)
endif
```

6.3.8. Type Extension Section

Symbol files also play an important role in the implementation of a new technique for checking subtype compatibilities, dynamically, for example being able to check the predicate

```
ExTypeOf (object, Type) : boolean
        ==      TRUE if Typeof(Object) is an extension of Type,
                FALSE otherwise
```

Wirth proposes a technique for implementing such type tests that requires a run-time linear search of the type hierarchy.

Our technique uses symbol files to record local type hierarchies of a component. The CL and PL compilers merge the local type hierarchies of read symbol files, and output the merged hierarchy in their own symbol file. In this way the type hierarchy is incrementally built and cascaded up the configuration hierarchy. During node compilation, a complete type hierarchy exists for the node. This type hierarchy is then labelled and the labels used to generate data that can be used to perform type tests in constant time.

The following example will serve to demonstrate the technique,

Given the type hierarchies:

```
define X: A,B,C
        type A=record ... end;  B=record (A) ... end;  C=record (A) ... end;
end.
```

and component no: X=1; type structure keys: A=9, B=27, C=33 we have

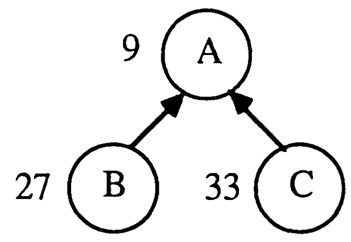


Fig 6.5 Type Hierarchy for X

The symbol file for X would have the following type extension section

Subtype		Supertype	
1	27	1	9
1	33	1	9
1	9	0	0

where

Type Extension Section
Extension *
Null

and

Extension
Subtype Owner : Integer
Subtype Key : Integer
Supertype Owner : Integer
Supertype Key : Integer

Note that no information needs to be recorded on the types themselves, only information on their subtype relationships needs to be recorded. A supertype with component number 0 and key 0 is used to terminate a hierarchy.

It is important that new extensions can be defined in other components, for example given

```
define Y: D,E;
        use X:A;
        type D=record (A) ... end;  E=record (D) ... end;
end.
```

and component nos: Y=1, X=2; type structure keys: D=18, E=41 we have

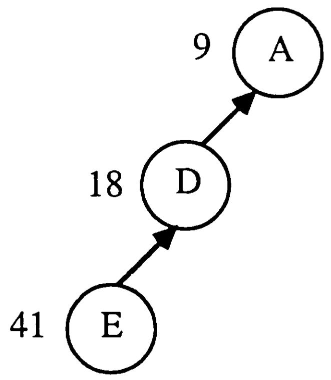


Fig 6.6 Type Hierarchy for Y

The symbol file for Y would hold the following table of extensions

Subtype		Supertype	
1	41	1	18
1	18	2	9
1	9	0	0

Plus the non-directly used extensions of X

2	27	2	9
2	33	2	9

We also define a further set of types as below:

```
define Z: F,G,H;
        use Y:D;
        use X:C;
        type F=record (D) ... end;  G=record (D) ... end;  H=record (C) ... end;
end.
```

and component nos: Z=1, Y=2, X=3, type structure keys: F=72, G=31, H=18 we have:

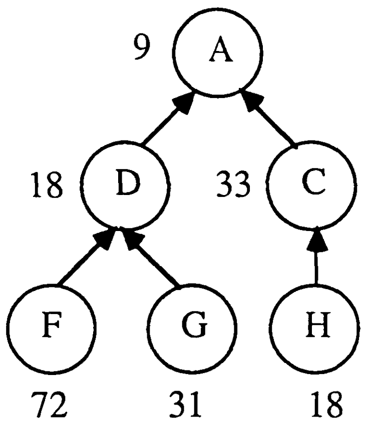


Fig 6.7 Type Hierarchy for Z

The symbol file for Y holds the following table of extensions

Subtype		Supertype	
1	72	2	18
1	31	2	18
1	18	3	33
2	18	3	9
3	33	3	9
3	9	0	0

Plus the non-directly used extensions of Y

2	41	2	18

Plus the non-directly used extensions of X

3	27	3	9

The merging procedure implied above is repeated until node compilation. Although we have shown the full type hierarchies in this example, in practice, compilers need only record a new branch of the type hierarchy if an explicit type test is made in the component.

During node compilation the merged type hierarchy is labelled with a pair of tags, the first tag specifies a type number for the type, the second tag specifies the highest tag in the type's subtype hierarchy. An inorder traversal of the type hierarchy is employed to label the type hierarchy as follows:

```
function label_type (P:Type; var tag:integer):integer;
var highest_tag:integer;
begin
        tag:=tag+1;
        P^.tag:=tag;

        highest_tag:=tag;
        foreach subType of P do
                let S=subType;
                highest_tag:=highest_tag max label _type (S,tag);
        end;
        P^.maxtag:=highest_tag;
        return highest_tag;
    end;
```

For the example the labelled hierarchy would be:

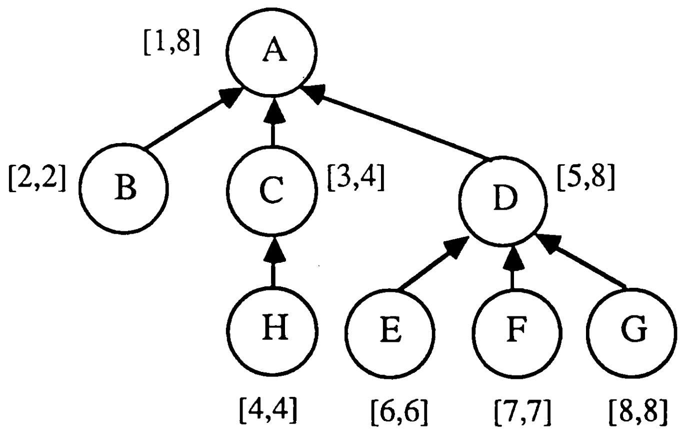


Fig 6.8 Labelled Type Hierarchy for X, Y, Z

where the types are labelled with [tag, maxTag]

In order to check that an object is an extension of a type X, the type test merely needs to check that the tag of the object is within the pair of tags for the type, namely:

Tag(Object) >= X.tag and tag(Object) <= X.maxTag

To check for a proper subtype, the check is:

Tag(Object) > X.tag and tag(Object) <= X.maxTag

To check for exact type equality the test is:

Tag(Object) = X.tag

Object tags can be set with the assignment:

Tag(Object) := X.tag

The number of extensions that a type has is:

X.maxTag - X.tag

6.3.8.1. Tag Identification

Since the extent of the type hierarchy is not known when a component is compiled, it must be possible for separately compiled components to be able to address Type tags at the time of compilation. Because types are uniquely keyed within a component, and components are uniquely identified by a timestamp, a component can safely address type tag data by referencing external tag identifiers that include these attributes, for example:

TYPE_<timestamp>_<id no>

For each node, the CL compiler then generates a tag identifier entry to satisfy the references made from the constituent components.

For the example, given:

TimeStamp(X)=891025132232
TimeStamp(Y)=891025184330
TimeStamp(Z)=891026101938

the CL would generate the following tag identifier data for link-loading:

Tag Identifier	Tag	MaxTag	
TYPE_891025132232_9	1	8	{A}
TYPE_891025132232_27	2	2	{B}
TYPE_891025132232_33	3	4	{C}
TYPE_891025184330_18	5	8	{D}
TYPE_891025184330_41	6	6	{E}
TYPE_891026101938_72	7	7	{F}
TYPE_891026101938_31	8	8	{G}
TYPE_891026101938_18	4	4	{H}

In addition to tag identifier generation, the CL compiler could be extended to generate type specific data, for example, a tag indexed table holding the size of types which can used by memory allocator/deallocator libraries.

SIZE [1]	<size_of_A>
SIZE [2]	<size_of_B>
SIZE [3]	<size_of_C>
SIZE [4]	<size_of_H>
SIZE [5]	<size_of_D>
SIZE [6]	<size_of_E>
SIZE [7]	<size_of_F>
SIZE [8]	<size_of_G>

6.3.9. Debugging Section

The debugging section is optional and can be used by compilers to record additional for debugging purposes. Since the information in the debugging section is not needed during compilation, and can be lengthy, it is placed at the end of the symbol file, alleviating compilers from having to skip over it. The Conic PL compiler for example, uses the debugging section to record the variable identifiers declared within a process.

6.3.10. Additional Facilities

A directory search capability is provided for the location for the symbol files. By default if a symbol file is not found in the current directory it is searched for in a set of standard directories. This allows users to override standard implementations of modules with their own versions, provided their is no inconsistent usage in the application. The search path can also be specified by users for greater control, and is useful where whole sets of symbol files are to moved within the file system or to different host file system. Symbol files are not currently shareable for heterogeneous targets. This is for two main reasons. Firstly, the size of types for each machine may be different, for example, integers may be 2 bytes on one machine 4 bytes on another. Secondly, our system currently supports conditional compilation of source files, allowing machine-dependent compilation of parts of a component, and this sometimes leads to types, and configuration specifications tailored to a particular machine. Currently multiple symbol files have to be built, one for each variant machine type.

Because symbol files hold useful information but in binary form, a utility called **show** is provided that displays the information in symbol files in a human-readable form.

6.4. CODE GENERATION

The basic code generation strategy of the CL compiler is to produce an implementation module for each group module. For portability the generated code is emitted in a high-level language rather than in assembly or machine language.

The basic outline of the implementation module generated for a group is:

```
module group

        function create_group(parameters):group_ptr;
                {sets up the group instance hierarchy for the group}
                {described in section 7.2 }
        end

        procedure link_group(G:group_ptr; parameters);
                {sets up a flat interconnection network for a group instance hierarchy}
                {described in section 7.2 }
        end

        function query_group (parameters):query_ptr
                {generates the configuration structure of a group}
                {described in 7.6}
        end

        type extension data
                {data for PL compilers to perform constant time subtype tests}
                {described in section 6.3.8. }
```

```
endmodule
```

The group elaboration procedures create_*group* and link_*group* are used to create and bind new instances of the group. The query_*group* function generates the configuration structure of a group instance. For nodes two extra procedures are also emitted:

```
procedure elaborate_node;
        link_node ( create_node(Unix_arguments),Unix_arguments)
        {Converts Unix arguments and calls the top-level group procedures}
        {described in section 7.1}
end

procedure node_interface;
        {Makes calls that pass back descriptions of node's ports}
        {described in section 7.4.1}
end
```

The first is a environment hook that retrieves the Unix arguments for the node process and calls the elaboration procedures for the node group. The second is used when dynamically binding nodes to retrieve information on the types of node ports and also to convert incoming messages from heterogeneous machines.

In order to generate these procedures in a single pass without forward references, the compiler builds and keeps configuration information until the end of the group module is reached. Only if no errors are found is the code emitted. By deferring code generation until the end of compilation, gives scope for additional optimisations.

6.5. LINK-LOADING

The CL compiler produces a linker file for each node. This file lists the names of all component object files needed to execute the node. The linker file is generated during node compilation by scanning the component table for the directory names of executable components. The CL compiler then attempts to open for each executable component the object file:

directory name / component name.o

If this fails, a search is made for the component, first in the current directory, and then in a set of standard directories. It is also possible for users to specify a set of directories for the CL compiler to search. If the object file is not found, a warning is generated. There is scope for potential inconsistency at this point, since the scheme for checking consistency of components is based only on symbol files, checks are also needed to ensure that referenced object files are also consistent.

This can be done by recording component timestamps within the object file and checking them against those present in the node's symbol table [Mitchell79]. An alternative is to defer consistency checking until node elaboration time.

Once the linker file is written, it is passed to the Unix link-loader along with the names of some run-time libraries, to generate a runnable node.

Runnable nodes for Unix have the same format as other executable Unix programs. Nodes can thus be used and manipulated like other Unix programs. For example, nodes can be renamed and used like other programs. No special command is required to run nodes.

6.6. PROJECT MANAGEMENT SUPPORT

A consistency management tool such as Make [Feldman79] on Unix is useful for program development, as it helps ensure that changes to components cause the recompilation of dependent components. Since writing Makefiles by hand is error-prone and tedious [Walden84], a tool called **ma** is provided that automatically generates a 'Makefile' for a component. **Ma** works by scanning the source of the component, along with the sources of all directly and indirectly used components to form a 'use' dependency graph of components. This graph is then translated into Makefile format. The Makefile produced by **ma** also adds rules that allow components to compiled multiple machine types. More sophisticated consistency management techniques such as smart recompilation are an obvious enhancement [Tichy86, Schwanke88] that could be used by the configuration and programming language compilers in conjunction with Make.

6.7. CHAPTER SUMMARY

This chapter has described the techniques used to compile configuration specifications into executable objects. Principal among these is the use of a new compact symbol file design. Symbol files are designed to (i) preserve strong type checking and visibility rules across separate compilations, (ii) provide earlier detection of inconsistent component usage, (iii) track the location of associated object files in the host file system, (iv) perform compile-time determination of the type hierarchy within nodes and (v) record additional debugging information in a fast, simple and scalable way.

A number of additional tools are provided: **ma** which produces Makefiles from available component source files, **show** which displays the contents of symbol files in human readable for, **pm** which using debugging information written in process component symbol files to list in source-level style the values of variable of crashed processes.

For each group module code is generated to create and bind new instances of that group and to query the configuration structure of the group. Data is also emitted for PL compilers to perform subtype tests in constant time.

The next chapter continues the description of the implementation model by describing the techniques used to elaborate, distribute, configure and query nodes.

Chapter Seven Run-time Techniques

In this chapter the techniques used to elaborate, distribute, and configure Conic nodes within the Conic environment are presented.

7.1. INTRODUCTION

Given a set of compiled nodes, techniques are needed for elaborating the configuration structure within nodes, for distributing nodes to remote machines, for binding distributed nodes together, and for reconfigure running nodes. Ideally these techniques should be fast, simple and scalable.

In our approach nodes are dynamically elaborated at run-time. This speeds up development times considerably and also saves on the file space that would be needed to hold elaboration data. Dynamic elaboration is particularly useful where nodes are parameterised, and the parameters are used to control the size or topology within the node, or where the code used to elaborate particular groups is to be invoked by process components. In approaches that statically elaborate configurations, these advantages are lost.

Support is also provided to distribute nodes to remote Unix machines. In order to overcome the short-comings of the remote execution facilities available under Unix, special nodes, called virtual targets (vt's) are provided which can be left as daemons on those machines at which new nodes are to be created. An interactive and extended version of configuration language **create** declaration can then be used to instruct the virtual target to create a new local node instance by-passing the normal Unix verification procedures.

Because we allow nodes to be independently developed and dynamically bound into a running program, support needs to provided to ensure that the interaction safety mechanisms of the language are enforced. This is achieved through the use of type descriptors called canonical data representations (CDRs) that are automatically generated by the configuration language compiler for use in checking the safety of node interactions.

CDRs are held within nodes along with the names of node ports and node port types. Such data can be accessed by configuration managers through a set of standard ports provided for each node. In order for a configuration manager to bind to these ports, it must find out the "internet" address of the node. This is achieved by querying a global name server that holds the names of all nodes in all programs. Each node periodically (every 10 seconds) reports its name and address to this server. If the server crashes, the information is automatically recovered by restarting a new server at the same address. Thus programs can continue running in the presence of server crashes. Once the address of a node is known to a configuration manager, it can interrogate the node for the names, types, and CDRs of its ports, and use the information provided to bind (and unbind) nodes together.

Support is also provided to allow tools to retrieve the actual configuration hierarchy built during elaboration. This information is be used, for example, by a graphical configuration manager to view and manage distributed program structures.

7.2. NODE ELABORATION

A distributed program specified using the Conic configuration language can have a very large and complex structure. Such programs are hard to efficiently execute within a heterogeneous distributed environment without transforming the program's configuration structure into a simpler, more efficient one.

A transformational approach, that maps the hierarchic topology of Conic configuration structures into a flat, non-hierarchic topology of interconnected primitive processes at run-time is adopted. An alternative to this approach would be to omit the transformation and use hierarchic configuration structures directly for inter-process communication. This implies a traversal by the run-time system of the interconnection path from the sending process to each corresponding interacting process. When both interacting processes are within the same group the interconnection path is of length 1. If one process is in group that is the parent of the other, then the interconnection path is of length 2. In general the path length is equal to the number of group module boundaries crossed by the traversal.

By flattening the group hierarchy within a node we reduce the interconnection path to length 1 for all intra-node communications within a node and to length 3 for all inter-node communications, an important improvement in efficiency.

The use of flat process topology within nodes also simplifies the data structures maintain by the underlying communication system.

7.2.1. Node Elaboration Procedures

For each group the CL compiler generates two procedures to elaborate the group:

- a create_*group* function that takes the formal parameters of the group and returns a pointer to a group instance data structure. This structures holds pointers to each sub-instance, and information on the hierarchic links of the group. The pointers are set by calling (i) for each sub-group instance, its corresponding create_*group* function, and (ii) for each sub-process instance a kernel function to create the task instance.
- a link_*group* procedure that takes a pointer to the group instance data structure generated by create_*group* and generates the flat interconnections for the processes within the group. For this the node is treated as a process in order to generate flat interconnections to and from the node.

The following example illustrates the results after these procedures a little further.

Given a node:

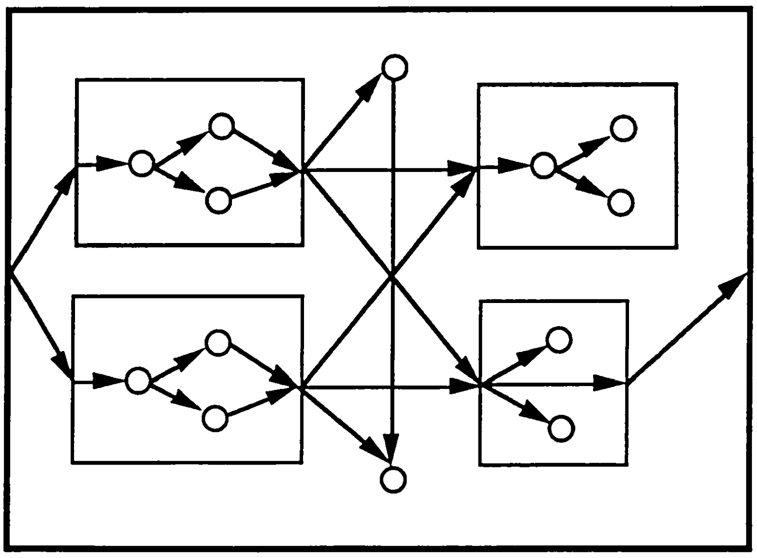


Fig 7.1 Example Node for Elaboration

After the create_*group* function we have:

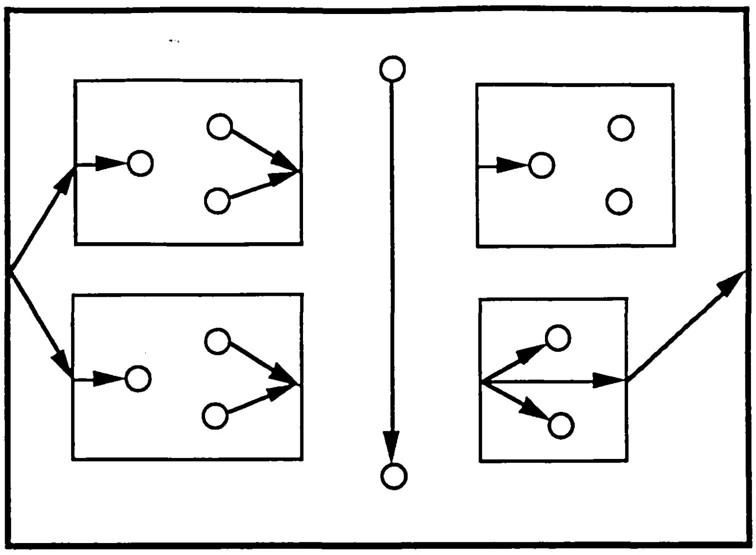


Fig 7.2 Node after the Create_*Group* Function

And after a subsequent link_*group* procedure we have:

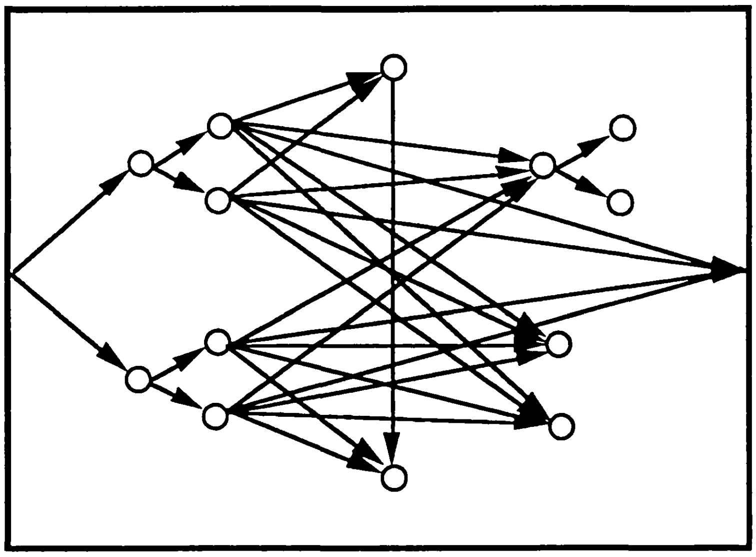


Fig 7.3 Node after the Link_*Group* Procedure

For nodes an additional procedure, elaborate_node is called by the node kernel to elaborate the node. The procedure needs to converts any Unix arguments supplied to actual group parameters.

```
procedure elaborate_node;
        link_node ( create_node(get_unix_arguments), get_unix_arguments)
end
```

The complete elaboration of any group can be made dynamically from within a running node by making the following procedure call:

link_*group* (create_*group* (parameters), parameters)

7.2.1.1. The create_*group* function

Generated create_*group* functions take the following general form:

```
function create_group ( formal parameters of group ) : group_ptr;
var G : group_ptr;
begin
        G := newgroup ( no. of instances of group, no. of ports of group );
        if group is a node then create_node ( no. of ports of group ) endif;

        foreach instance I declared in group do
                if I is a group then
                        G^.Inst [ ord I ] := create_I ( actual parameters of I )
                else { I is a process }
                        G^.Inst [ ord I ] := create_task ( I , actual parameters of I );
                endif
        endfor

        foreach hierarchic link X to Y.Z or Y.Z to X do
                G^.Port [ ord X ]:=add_link ( G, ord Y, ord Z );  { -- Saves data in G }
        endfor

        return G
end
```

The newgroup function allocates memory for the group instance data structure. This structure consists of an instance table for the sub-instances of the group and a port table form holding information on the hierarchic links made to the port. Each sub-instance and port of the group is given an ordinal value by the compiler to uniquely index these tables.

The instance table is set by calling (i) for each sub-group instance, its corresponding create_*group* function, (ii) for each sub-process instance the kernel function create_task to create the task instance. The create_task function returns a unique process number for each created task instance within the node. Process number 0 is used to designate node ports.

Hierarchic link information is saved in a port table. Each entry consists of a list of linked tuples of the form <instance no, port no>. An instance number of zero is used to signify a forwarded link.

If the group is a node an additional kernel call is made to inform the kernel of the number of node ports to cater form.

7.2.1.2. The link_*group* procedure

Generated link_*group* procedures take the following general form:

```
procedure link_group ( G:group_ptr; formal parameters of group );
begin
```

```
        foreach non-hierarchic link W.X to Y.Z do
            walk_link ( port (G, W, X ) , port (G, Y, Z ) )
        endfor

        foreach subgroup instance I of group do
            link_I ( G^.Inst [ord I ], actual parameters of I );
        endfor
    end
```

The port function constructs a port descriptor from the specified instance number, port number. The descriptor holds information on the type of instance (group or task), its process number if a task instance, and the type port (exitport or entryport).

The walk_link procedure generates all possible flat links between two linked ports. The link_*group* procedure must also call the corresponding link_*group* procedure of each sub-group instance.

The walk_link procedure takes a link between two non-hierarchic ports and generates calls of all the possible flat interconnections. This is done by locating each task exitport connected to the supplied exitport, and linking it each task entryport connected to the supplied entryport.

```
    procedure walk_link (XP, EP)
    { Expand group exitports }
    begin
        if XP is a task_port then
            walk_eport (XP,EP)
        else { XP is a group port }
            foreach internal port IP linked hierarchically up to XP do
                walk_link (IP, EP)
            endfor
        endif
    end
```

Whenever an exitport is hierarchically linked to a sub-group exitport, the walk_link procedure is recursively invoked with the sub-group exitport as the parameter. Whenever the exitport is hierarchically linked to a sub-task exitport, the walk_eport procedure is invoked to expand the entryport side of the link. In this way all described task-to-task links can be generated.

The walk_eport procedure is similar to the walk_link procedure except that on encountering a task entryport a kernel call is made to generate the flat link.

```
    procedure walk_eport (XP, EP)
    { Expand group entryports }
    begin
        {assert XP is a task exitport }
        if EP is a task port then
            do_flat_link (XP, EP)                { -- Kernel routine }
        else { EP is a group port }
```

```
                foreach internal port IP linked hierarchically down from EP do
                    walk_cport (XP, IP)
                endfor
            endif
        end
```

Hierarchic links to a node port are also flattened in this way, with the node considered a task instance with process number 0.

7.2.2. Performance

The table below presents times for elaborating Batcher's bitonic sorter example from Chapter 4. The times were measured on a lightly loaded SUN 3/60 workstation using the Unix ftime() procedure. Each case was run 3 times. The worst time measured is used in the table.

Elements	Tasks	Links	Time (secs)
2	13	15	0.040
4	18	27	0.060
8	36	67	0.100
16	92	187	0.280
32	252	523	0.700
64	684	1419	1.880
128	1804	3723	5.280
256	4620	9483	12.470

Fig 7.4 Node Elaboration Figures for Batcher's Bitonic Sort

Elements is number of elements to be sorted, and is passed to the node as a parameter when the node is executed. Tasks is the number of primitive processes created. For each case, Tasks includes 12 executive tasks. Links is the number of flat process-to-process links established. For each case, Links includes 11 executive links. Default links are not included in the Links figures. Time is the time taken to elaborate the node and generate a flat interconnection structure. Time includes the time taken by the Conic run-time kernel to allocate memory for each task, as well as the time taken to initialise kernel data structures. Time also includes the time taken to perform default links for each task.

7.3. NODE CREATION

Nodes can be created in one of two ways. Firstly since nodes are treated uniformly like other compiled programs on Unix, they can always be directly executable. For remote Unix execution, direct execution can be cumbersome and slow, requiring logging into each host in turn in order to run a node. The Unix remote command invocation facility, rsh, can be several orders of magnitude slower than local invocation [Bhattacharyya88]. In order to speed remote node creations, special nodes called virtual targets (vt) provided. These also increase the flexibility of the system by allowing resources for the node, such as a display windows to be easily created.

7.3.1. Direct Creation

Nodes compiled for Unix hosts can be distributed and execution started, by logging onto the desired host, and executing the command:

node parameters - instance system

where *node* is the name of the link-loaded node file, *parameters* are zero or more actual parameters, *instance* is the instance name for the node (if omitted this defaults to the process-id under Unix and the node name on targets), *system* is the name of the system under which the node is to be registered (if omitted this defaults to the user's login name). System can be treated like a name for the program.

The parameters supplied are passed as actual parameters to the node's configuration specification.

7.3.2. Indirect Execution via VT's

Alternatively, nodes can be created via special **vt** nodes. Vt's provide a Conic message passing interface for receiving Unix commands from remote machines and executing them. Once started, vt's can be left to run in isolation, as daemon shells on the remote host. Vt 's expect create messages of the form:

"node parameters - instance system"

from remote nodes, in particular from the Conic configuration managers **iman** and **ConicDraw**. When a vt is started without parameters all received create requests are executed directly. When a vt is started with parameters, the parameters are assumed to name a Unix command to prefix before all received create requests. By supplying parameters to the vt, it is possible to create additional resources or context for each created node. For example, to create an X windows [Scheifler86] for each node, or to run each created node under the Unix debugger adb, we can start the following vt 's:

```
vt xterm -e - window mysystem
vt adb - debug yoursystem
```

The parameters of a vt normally name a Unix shell script that performs more complex resource allocation operations prior to node execution.

Vt 's are distributed and started like other nodes (ie. directly by logging into the remote host, or via other vt's), for example

> **vt** *vtparameters - vtinstance system*

Once a create message is received, vt's construct a new Unix command by prefixing their own parameters before the create request, i.e.

> *vtparameters* node parameters - instance system

This command is then executed using the 'exec' system call.

Node creation messages can be sent to vt's with the iman command

> **create** *instance node parameters* **at** *vtinstance*

The use of virtual target names instead of actual machine names also means that node creation scripts are machine independent. Executing the same sequence of node creation commands at different times may result in nodes running on different machines, since virtual targets can be removed and recreated elsewhere.

7.3.3. Target Creation

To create a node on a target, the download command **dl** can be used, for example;

> **dl** *target node parameters - instance system*

or if a downloading vt is running, i.e: if the vt was started with a command of the form:

> **vt dl** *target - vtinstance system*

the configuration manager command:

> **create** *instance node parameters* **at** *vtinstance*

can be used instead.

In addition to the object code and initialised data for the node, the downloader also sets up a Unix-like process stack, ie. setting up **argv**, **argp**, and **envp**. This involves downloading the node arguments and a set of environment variables to the target. The use of Unix-like process environment allows many parts of the Conic run-time system to remain common to hosts and targets.

7.4. NODE BINDING

Provided the message types of the two ports are compatible, node ports can be bound with the interactive configuration manager command:

link *node.exitport* **to** *node.entryport*

The choice of Pascal's name equivalence semantic to check intra-node binding is difficult to enforce when nodes may be developed and run independently on heterogeneous hosts. An implementation would require distribution and sharing of symbol files across the hosts. Therefore strict name equivalence was dropped in favour of weaker structural type checking, implemented by generating Canonical Data Representations for message types.

7.4.1. Canonical Data Representation (CDR)

Canonical data representations are used for validating node binding commands, for transformation of messages across heterogeneous boundaries, and for monitoring messages sent from a node.

For each node the CL compiler generates a node_interface procedure makes kernel calls to set up the CDRs for the message types of each node port.

```
procedure node_interface;
        {Makes kernel calls that describe the node ports}
        foreach nodeport do
            nodeport (porttype, portname, requestCDR, replyCDR)
        end
end
```

CDRs are strings which are generated by unrolling the type structure of a message type and are formed as follows:

1) Standard types are represented by a single lowercase letter: integer by 'i', real by 'r', natural by 'n', longint by 'l', char by 'c', boolean and byte by 'b'. If a standard type can be packed and occurs within a packed type, uppercase is used.

2) Subranges are represented by the appropriate base type representation.

3) Enumerated types are represented by 'b' if the cardinality of the type is > 255,

otherwise by 'i'.

4) Arrays are represented by the an integer representing the number of elements in the array, followed by the representation of the element type in parenthesis. Parenthesis are normally omitted if the element type is not of set type, array type or record type.

5) Records are represented by the conjunction of the representations of their fields.

6) Sets are represented by an integer representing the set size, followed by the letter 's'

For example the type

```
record
        low : integer;
        value : real;
        name : packed array [5..10] of char;
        dates : array [1..10] of record
                year : 1900...2001;
                month : (jan, feb, mar, apr,may, jun, jul, aug, sep, oct, nov,dec);
                day : 1..31;
                end;
        capabilities : array [1..99,1..4] of set of char;
end;
```

is described by the CDR "ir6C10(ibb)99(4(256s))"

For stronger checking, greater type discrimination could easily be added, by explicitly adding special letters for enumerated types, subrange types, and record types.

7.4.2. Node binding compatibility

To check that two message types are compatible, the CDR strings are checked for string equivalence. For extended types the CDR of the supertype needs to be a leading substring of the CDR of the subtype.

For increased speed hashed CDRs could be used to check binding [Scott88], at the cost of occasional invalid binding. Hashed CDRs are more useful where the type checks need to made for each message received. In our system, checks are only required on binding, which is relatively infrequent compared to the number of messages subsequently transmitted on the bound ports.

7.4.3. Heterogeneous Message Passing

CDRs are also used by the run-time system to convert incoming messages from dissimilar machines into local format, for example a real value sent from a VAX is converted automatically to its equivalent bit pattern on a Motorola 68000. Since CDRs are held at both the sending node and the receiving node there is no need to send CDRs as a part of a message.

To offset the cost of including conversion code of each machine type, run-time systems could convert outgoing messages into some standard format, such as ASN.1 [ISO85]. This results in two message conversions per message transmission, even if machine types are the same, but does mean that future machine types can be accommodated without having to write new conversion routines for the machine type.

Machine types could be extended to include a sub-field for indicating the programming language used to program the component. This field could then be used to initiate inter-language data type conversions, for example, converting matrices held column-wise in Fortran, into the row-wise matrices of another language.

7.5. NODE CONFIGURATION

Two interactive configuration managers are currently provided for node configuration. Iman which provides a textual interface and ConicDraw which provides a graphical interface. The principal configuration commands accepted by Iman are:

manage system { switches context to system }

create *instance type* [*parameters*] **at** *VTinstance*

remove *instance*

link *instance.exitport* **to** *instance.entryport*

unlink *instance.exitport* **from** *instance.entryport*

Given virtual targets alpha, beta, gamma, delta and epsilon running on various machines, the dining philosophers system

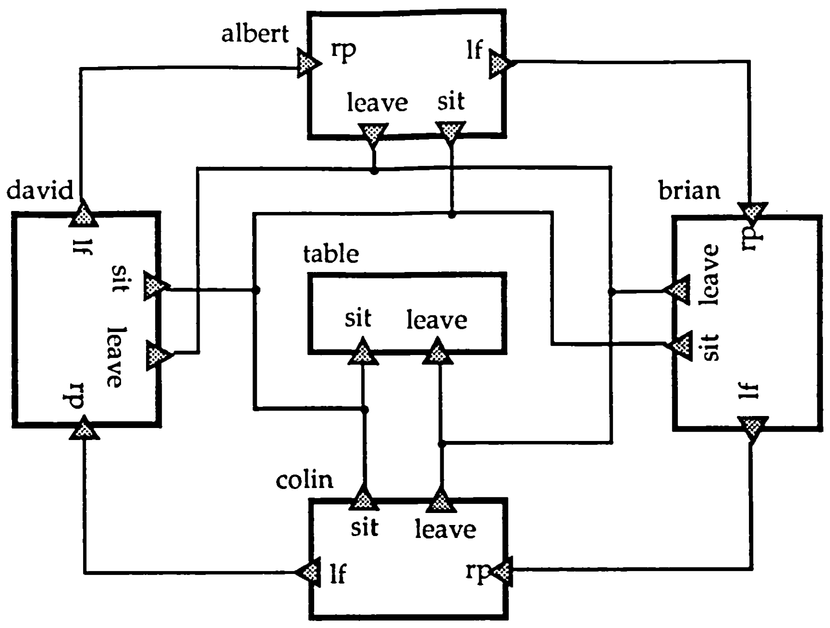


Fig 7.5 System of Dining Philosophers

can be distributed and configured as follows:

```
create albert philosopher at alpha
create brian philosopher at beta
create colin philosopher at gamma
create david philosopher at delta
create table table at epsilon
link albert.lf to brian.rp     link albert.leave to table.leave   link albert.sit to table.sit
link brian.lf to colin.rp      link brian.leave to table.leave    link brian.sit to table.sit
link colin.lf to david.rp      link colin.leave to table.leave    link colin.sit to table.sit
link david.lf to albert.rp     link david.leave to table.leave    link david.sit to table.sit
start albert  start brian  start colin  start david
```

The **start** command sends a message to a node, informing it that it has been configured. This message can be received by any or all the processes within a node and for example, used by them to begin interaction. A complementary command called **stop** is also available which sends a message to a node informing informing it that it is to be reconfigured. Again this can be received by any or all the processes within a node and for example, used by them to suspend interactions. A more declarative and rigorous approach to change management is described in [Kramer88].

In addition to performing node configuration commands, Iman allows queries to made on the current configuration state of a program using the commands:

nodes	{ lists the node instances within managed system }
llnodes	{ provides a more detailed listing than nodes }
systems	{ lists names of all running systems }

ports *instance* { lists ports of *instance* }

llinks *instance* { lists links of *instance* }

ConicDraw is similar to Iman but commands are performed graphically [Kramer89]. Linking, for example, is accomplished by drawing a line from the exitport of one node to the entryport of another. Currently the structure of running programs is displayed on, and manipulated from, Apple Macintosh computers, e.g.

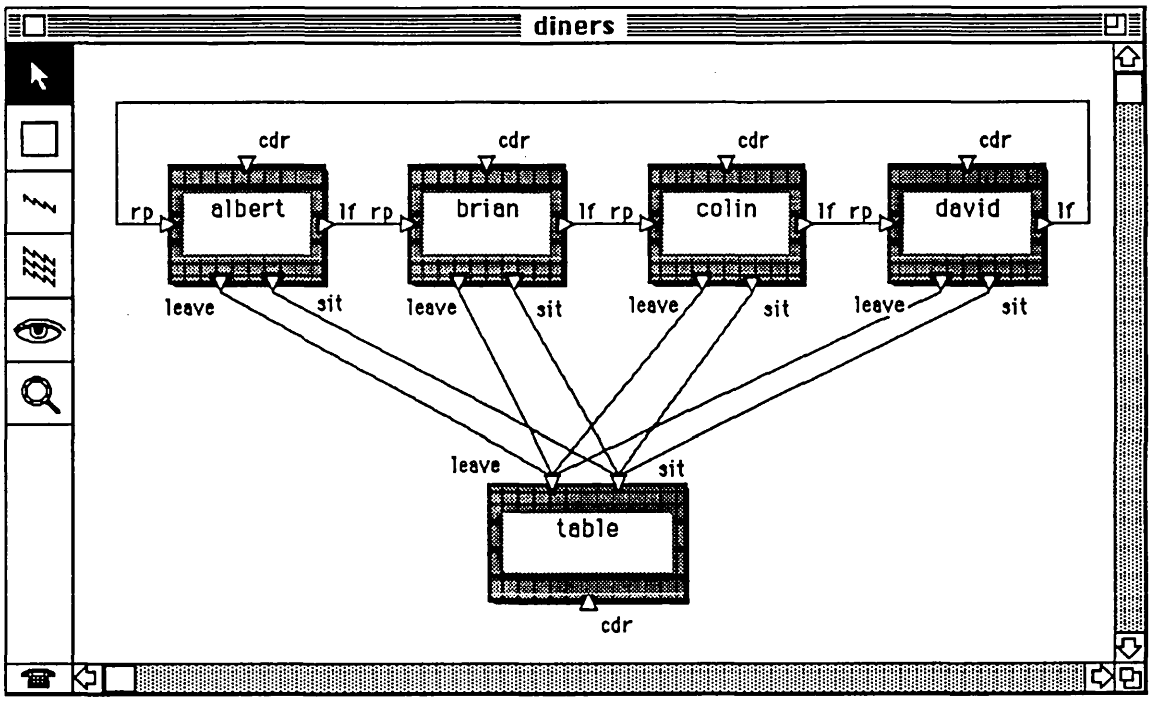


Fig 7.6 Example of a ConicDraw Window

ConicDraw interacts with nodes through a specially modified version of Iman called Gman, to which it linked over a serial link.

7.6. QUERYING NODE STRUCTURE

The internal details and structure of a node can also be queried at run-time. Requests for the internal structure of a node are answered by a special process in the executive called **structman**. This holds a representation of the current configuration hierarchy for the node. Procedures to produce the hierarchy are generated by the Configuration Language compiler in an analogous way to the procedures for node elaboration.

Replies from structman return the structure of the node in a coded ascii format which is designed for compactness. These replies are used by ConicDraw when displaying the internal hierarchy of a node. An interesting addition to ascii representation would be to return structural data as Prolog clauses. These could then be used directly by analysis tools written in Prolog.

7.7. NODE DEBUGGING

Several tools are provided to help in debugging nodes. A tool for viewing the variables of crashed processes, a tool for viewing the source lines executed by processes within a node, and a tool for viewing the messages sent between nodes.

7.7.1. Post-Mortem Dumps

When a process crashes, or under program control, a dump of the process memory is performed by the run-time executive. A tool, called **pm** is provided which when invoked reads the debugging section of the process symbol file, and provides a listing of the contents of the process memory in high level form, e.g:

```
Node alpha, Instance 23, module=t3, clock=2000, failed at line 337 because: task completed

VARIABLES:-
a1= ( -1000, -900, -800, -700, -600, -500, -400, -300, -200, -100, 0, 100, 200, 300, 400, 500, 600, 700,
 800, 900, 1000 );
c= ( 'a', 'b', 'c', '2', '7', ' ', 'a', ' ', 'x', 'y', '0', '6', '3', '0', '0', '0', ' ', 0#C, 0#C, 0#C);
c2='z';
colors=[ green..orange ];
cset=[ '('..'+' '-'..':' 'a'..'{' '}' ];
head= HEAP (251588700) ^
 RECORD
  val=200 |
  next= HEAP (251588694) ^
   RECORD
    val=199 |
    next= NIL
   END
 END;
input=File not Open;
k=3;
letters=[ '0'..'9' 'a'..'z' ];
r= 2.029297e+00
```

Fig 7.7 Example of a Post Mortem Dump

7.7.2. Node Playback

The processes within a node can be specially compiled to log to a tracefile information on their name and the source line number of each line executed. A tool called **pb** is also provided which when called reads the tracefile, and plays back in separate windows the lines logged by the processes.

```
--------rcv.tas  7--------------kernel.tas  1---------------snd.tas  6--------
|  15   i, j, k:integer; || 309 (procedure joinkid(||  15   i, j, k:integer; |
|  16 begin              || 310 procedure joinkid; ||  16 begin              |
|  17   i := 0; j := 0; k|| 311 var p, q: PaidLst; ||  17   i := 0; j := 0; k|
|  18   writeln(' ':30, '|| 312 begin              ||  18   writeln('Hello, S|
|> 19   flush(output);   ||>313   if (oldk <> newk)||  19   flush(output);   |
|  20   i := i + 1;      || 314     p := kidmap[old||  20   i := i + 1;      |
|  21   action           || 315     while (p <> nil||█ 21   action           |
|  22     j := j + 1;    || 316       q := p; p := ||  22     j := j + 1;    |
|  23     action         || 317       putinkid(newk||  23     action         |
|  24       k := k + 1;  || 318     end;           ||  24       k := k + 1;  |
|  25       receive signa|| 319     pr := kidmap[ol||  25       send signal t|
|  26       commit;      || 320     while (pr <> ni||  26       commit;      |
|  27     end;           || 321       if (pr^.rely ||  27     end;           |
|  28     delay(1); abort|| 322         putinrel(ne||  28     commit;        |
|  29   end;             || 323       end          ||  29   end;             |
|  30   writeln(' ':30, '|| 324       else         ||  30   writeln('Snd, i:'|
|  31   delay(100);      || 325         rmfromdep(n||  31   delay(100);      |
|  32 end.               || 326       qr := pr; pr ||  32 end.               |
|  33                    || 327     end;           ||  33                    |
|  34                    || 328     pd := kidmap[ol||  34                    |
|  35                    || 329     while (pd <> ni||  35                    |
------------------------------------------------------------------------------
single step
```

Fig 7.8 Example of Node Playback

Pb accepts commands to play forward one line at a time, to play forward until the next process switch, and to play continuously forward. Pb can also play tracefiles backwards, either one line at a time, until the previous process switch or continuously until the first logged line. This is useful where a node has crashed and the programmer does not wish to play the entire execution from the beginning i.e he only wishes to know which lines were last executed.

7.7.3. Message Monitoring

The messages sent on any inter-node link can be intercepted and listed by a filter called **spy**. Spy takes the same arguments as the **link** command. When invoked spy relinks the two node ports concerned to itself.

For example, if before spying we have:

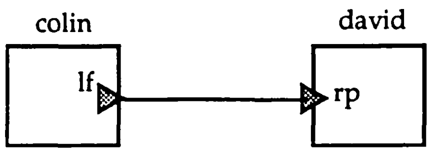


Fig 7.9 Inter-node Link prior to Spy

After the command **spy** colin.lf **to** david.rp, we would have:

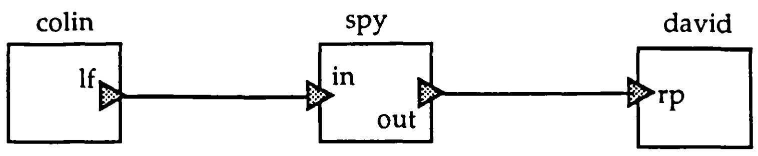


Fig 7.10 Inter-node Link with Spy Filter

Before performing relinking, spy also interrogates the sending node for the CDRs of its port's request and reply types. The basic action of spy is then to repeatedly:

```
receive msg from in;
use requestCDR to list msg in human readable form
send msg to out await reply -msg
use replyCDR to list reply-msg in human readable form
send reply-msg to in
```

The original link is restored by spy before it terminates.

7.7.4. Final Comments

The debugging tools currently available work in isolation, the next logical step would be integrate them with ConicDraw. Hopefully this would lead to a comprehensive debugging system for distributed programs.

Although these debugging tools are available we have found that the most effective debugging tool is still careful thought and analysis.

7.8. CHAPTER SUMMARY

This chapter has presented an algorithm for dynamically elaborating the configuration structures of the language. The elaboration algorithm generates and transforms the hierarchic instance and interaction topology of a node into a more efficient flat topology while maintaining the hierarchic representations. The hierarchic representation is used by a graphical configuration manager to view the configuration structure of nodes. Results showing the performance of the algorithm for elaborating the structure of Batcher's sorter have also been given.

The CL compiler supports interaction safety of independently developed components by generating canonical data representation (CDR) strings of declared node port types. CDR strings can be used to check that bound node ports are compatible. CDRs have also been profitably used for heterogeneous message conversions, and monitoring inter-node messages.

Tools have also been provided for querying the structure of nodes, and for debugging nodes. Existing techniques in the Conic toolkit for distributing, configuring, and viewing distributed programs structured with this language have also been outlined.

Chapter Eight Conclusion

8.1. SUMMARY OF WORK

The work described in this thesis was motivated by the inadequacy of existing languages for developing distributed programs. The approach advocated has been to provide a separate and specialised language, termed a configuration language for structuring distributed programs that is lucid, flexible and practical. This is used in conjunction with a language for programming, to provide a complete notation for writing distributed programs.

The configuration language designed has attempted to fulfil the key requirements for structuring distributed programs identified in Chapter 2 of the thesis. Principal among these was the need to specify distributed program structures separately from the algorithmic implementation of programs. This goal has been met by the provision of a completely separate language that incorporates a new program structuring mechanism called the group module. The mechanism is powerful and allows hierarchic program structures based on primitive concurrent processes to be expressed as well as replicated, variant, parameterised and recursive program structures. Specified program structures are both modular and reusable and can be used to build ever larger programs in a scalable way. The language is declarative and aims to be independent of particular programming languages. The language also supports type-safe composition of distributed program parts. The language has been efficiently implemented on an existing distributed system, and is designed to support modifications to the structure of running programs.

In addition many of the program structures described by the language can be represented graphically which can act as a further aid to readers.

The thesis has also presented an implementation model for implementing the configuration language. The model focuses on providing efficient and pragmatic solutions to support large-scale distributed development within an heterogeneous environment.

The compilation techniques presented employ a new symbol file design for efficiently representing the interfaces of separately compilable units. Symbol files are also profitably used in the development of a new technique for performing type extension checks in constant time rather than linear time. Symbol files are also used to track object files in the host file system. A tool has been provided to automatically generate Makefiles from component sources.

Unlike other systems, which elaborate program structures at link-load time, we present a new technique for elaborating program structures at execution time. This uses a recursive algorithm for transforming the hierarchic structures of group modules into a more efficient heterarchic structure for use by the run-time system.

Canonical data representations (CDRs) have been used to check that bound nodes interact safely, and allows independently developed nodes to be incrementally added to a running program. CDRs have also been profitably used for heterogeneous message conversions, and monitoring inter-node messages.

Tools have also been provided for querying the structure of nodes, and for debugging nodes.

Existing techniques in the Conic toolkit for distributing, configuring, and viewing distributed programs structured with this language have been also been outlined.

The configuration language and tools described have been implemented and are available as part of the Conic Toolkit which is use in institutions around the world. The configuration language is also being used used as the starting point within Esprit project 2080 for building reconfigurable and extensible parallel and distributed systems (REX).

8.2. CRITICAL EVALUATION

The configuration language shares similar aims to many existing languages for program structuring. It provides a very abstract program structuring mechanism that has much greater expressive power than hitherto provided by others.

The use of a declarative language has resulted in clearer program descriptions. Few concepts are needed to master the language. The language supports abstraction and information hiding. The language also encourages top-down design. The structures described form abstract types that can be reused in different contexts.

The language and structuring mechanism has also been implemented in more dynamic and flexible way. Configuration structures are compiled not interpreted, and elaborated at run-time rather than link-load time. At run-time an algorithm is provided to transform the recursive interaction topology of a program into a more efficient flat topology for use by the run-time system. The implementation also provides query support that allows tools to query the structure of running programs.

The granularity and nature of the atomic components for our configuration language has focussed the language firmly on the issues of combining programming-in-the-large with distributed programming. Because of their concurrent nature and by our use of typed message-passing ports, processes can be considered as abstract computers, and configuration structures as interconnected networks of abstract computers.

The configuration language could be improved in a number of areas. The ability to group together a collection of ports, and treat them as a structured port would be a useful addition. Such collections could be considered as port records or port sets. For example, a set of file ports could be described and used as follows:

```
define filesystem : fileinterface
        exitport     open: filename reply  filedescriptor
        exitport     close: filedescriptor  reply signaltype
        exitport     read: readrequest reply  buffer
        exitport     write: writerequest reply signaltype
        portset      fileinterface  = (open, close, read, write)
end

task module client
        use filesystem : fileinterface
        <rest>
end

task module server
        use filesystem : fileinterface (reversed)
        <rest>
end

group module example
        use filesystem : fileinterface (reversed)
        use client;  server;
        bind client.fileinterface to server.fileinterface
        bind server.fileinterface to  example.fileinterface
end
```

The **reversed** operator above, is used to reverse the directionality of ports ie.

change entryports into exitports and entryports into exitports. Like CCS, binding could be done on the basis of identically named ports of complementary port types. Set operators such as union and intersection could also included for constructing new portsets. A more radical proposal would be to omit port directionality and port typing from configuration specifications altogether, and relying on the compiler to infer types and directionality. This would make configuration specifications less verbose, but may also be lead to less clear specifications.

A major omission in the language is the lack of dynamic program structuring at all levels, for example, being able to describe and initiate possible changes to the topology of a group module. Ideally such changes should be specified at the configuration level.

A first attempt at expressing such changes, resulted in many new concepts, such as (i) unbounded port families, (ii) guards that intercept incoming messages and trigger local reconfigurations, (iii) schemes to generate new instance names and select old names, and (iv) exception clauses to activate when failures arose. Difficulties also arose in realising an efficient protocol for concurrently activated changes when changes were at different levels of the configuration hierarchy, distributed and caused interference. Because of these complexities and for pragmatic reasons, this solution was not pursued.

An important stumbling block was also that including the conditions for reconfiguration actions within the configuration language, led to a language approaching the power of a programming language. A way of simplifying the approach would have been to separate out the two areas of concern, (i) **what** the change is, and (ii) **when** the change is to be initiated, by making the configuration language responsible for describing what changes are possible, and leaving it to the programming language to control when those changes should be initiated.

A more radical alternative would be to consider if some form of lazily evaluation strategy for the configuration language would help.

8.3. SOME SUGGESTIONS FOR FUTURE WORK

The node model, although efficient and practical, is not as appealing as a fully distributed implementation of group structures. Such an implementation model would allow programs to have arbitrary configuration levels above the node level. In highly parallel computers such as hypercubes and transputer arrays, what is required is for the programmer to defer the selection of the mapping strategy until program startup time. Although the programmer could currently map each process into a separate node, the current implementation model does not provide for the hierarchic or recursive construction of node structures. A number of difficult problems need to be overcome however, for example, how and where is information on the structure of (recursively) distributed groups held, can such groups be efficiently managed and reconfigured, can a distributed implementation cope with failures and network partitioning, can reconfiguration actions be successfully serialised?

The ability to break the rigid modular hierarchy of group modules is sometimes desirable. Currently if a program needs to connect to an existing instance, e.g. a known service, such a running file manager the configuration programmer has to either (i) 'float up' the required ports from the client component to the client node and link these to the server components, or (ii) pass down to the client an instance parameter that identifies the server. In either case, a great deal of unnecessary and cumbersome configuration programming is needed at each configuration level above the client level.

A possibility would be to allow known instances to be directly specified within a configuration whenever required, for example with the declaration:

```
requires server:servertype;
```

These instances would be made known when created with for example, the declaration:

```
create server : servertype;
provides server;
```

Such extensions do imply the existence of a global namespace for heterarchically accessible instances. Rules and mechanisms for defining, creating, searching, and deleting namespaces, as well as a rule for resolving instance name-clashes are also required.

Another possibility would be to see if a wider spectrum of programming language components could be integrated, for example Prolog, ML and Smalltalk. The integration and interaction of the typing systems used in these language presents many interesting problems however.

Behavioural and other specifications, such as performance and fault-tolerant specifications could also be integrated. One way to tackle this would be by a family of interacting companion languages that can inherit the structural description of a program as framework for their own specifications. This suggests a ramified approach to distributed program specification with the configuration language sitting at the apex.

Finally the lack of a complete formal definition could be addressed.

8.4. FINAL REMARKS

The main contribution of this work has been to demonstrate the practicality and versatility of a dual language approach to distributed program development. The approach provides an abstract declarative language for structuring distributed programs as sets of hierarchically interconnected concurrently executable program parts. The language can be coupled with one or more programming languages to provide a complete programming system for distributed programs with a clear and manageable structure.

Perhaps the most pleasing results of this work has been the unexpected application areas that the language has been put to use by colleagues and students. The language has been used in the construction of distributed process control programs for small networks of machines controlling mining equipment, and for distributed servers, databases and games. The language has also been used for simulation studies, for writing protocols, for building an object-oriented management system, for writing multi-loop self-tuning adaptive controllers and for the implementation of parallel algorithms, for example, for the travelling salesman problem, fast-fourier transformations, image processing and neural networks. The entire Conic run-time and support system is also structured with the language.

References

[Abelson85] H. Abelson, G. J. Sussman and J. Sussman, "*The Structure and Interpretation of Computer Programs*", MIT Press, 1985.

[Ada83] "*Reference Manual for the Ada Programming Language*", ANSI/MIL-STD-1815A, American National Standards Institute, 1983.

[Agha86] Gul Agha, "*An Overview of Actor Languages*", ACM SIGPLAN Notices, Vol 21, No 10, October 1986, Pages 58-67.

[Albert88] E. Albert, K. Knobe, J. Lukas, and G. L. Steel Jr., "*Compiling Fortran 8x Array Features for the Connection Machine Computer*", In ACM SIGPLAN PPEALS 1988 Parallel Programming Experiences with Applications, Languages and Systems, New Haven, Connecticut, July 19-21, 1988, SIGPLAN Notices, Vol 23, No 9, September 1988, Pages 42-56.

[Andrews88] Gregory R. Andrews, Ronald A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend, "*An Overview of the SR Language and Implementation*", ACM TOPLAS, Vol 10, No 1, January 1988, Pages 156-177.

[Bach86] Maurice J. Bach, "*The Design of the UNIX Operating System*", Prentice-Hall International, 1986.

[Backus78] John Backus, "*Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*", CACM, Vol 21, No 8, August 1978, Pages 613-640.

[Barbacci88] Mario R. Barbacci, C. B. Weinstock, and J. M. Wing, "*Programming at the Processor-Memory-Switch Level*", In Proc. of the 10th Intl. Conf. on Software Engineering, Singapore, April 11-15, 1988. Pages 19-28.

[Batcher68] K. E. Batcher, "*Sorting Networks and their Applications*", In Proc. 1968 Spring Joint Computer Conf., AFIPS Press, 1968, Pages 307-314.

[Bate86] George Bate, "*Mascot 3, an informal introductory tutorial*", IEE Software Engineering Journal, Vol 1, No 3, May 1986, Pages 95-102.

[Bhattacharyya88] Mitali Bhattacharyya, David Cohrs and Barton Miller, "*A Visual Process Connector for Unix*", IEEE Software, Vol 5, No 4, Pages 43-50.

[Black87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy and L. Carter, "*Distribution and abstract data types in Emerald*", IEEE Trans. on Software Engineering, Vol SE-13, No 1, January 1987, Pages 65-76.

[Bron85] C. Bron, E. J. Dijkstra and T. J. Rossingh, "*A note on the checking of interfaces between separately compiled modules*", ACM SIGPLAN Notices, Vol 20, No 8, August 1985, Pages 60-63.

[Burstall84] Rod Burstall, "*Programming with Modules as Typed Functional Programming*", In Proceedings of the Intl. Conf. on Fifth Generation Computer Systems 1984, edited by ICOT, November 1984, Pages 103-112.

[Campbell86] R. H. Campbell, A. M. Koelmans and M. R. McLauclan, "*STRICT: a design language for strongly typed recursive circuits*", Tech. Rept. 211, Computer Laboratory, University of Newcastle upon Tyne, April 1986.

[Carriero89] Nicholas Carriero and David Gelernter, "*Linda in Context*", CACM , Vol 32, No 4, April 1986, Pages 444-458.

[Chandy88] K. Mani Chandy and Jayadev Misra, "*Parallel Program Design: A Foundation*", Addison-Wesley, 1988.

[Clocksin81] William F. Clocksin and Christopher S. Mellish, "*Programming in Prolog*", Springer-Verlag, 1981.

[Coulas87] Michael F. Coulas, Glenn H. MacEwen, and Genevieve Marquis, "*RNet: A Hard Real-Time Distribued Programming System*", IEEE Trans. on Computers., Vol 36, No 8, August 1987, Pages 917-937.

[DeRemer76] Frank DeRemer and Hans H. Kron, "*Programming-in-the-Large versus Programming-in-the-Small*", IEEE Trans. on Software Engineering , Vol 2, No 2, June 1976, Pages 80-86.

[Dijkstra68] E. W. Dijkstra, "*Co-operating Sequential Processes*", In Programming Languages, Editor F. Genuys, Academic Press, 1968, Pages 43-112.

[Dijkstra71] E. W. Dijkstra, "*Hierarchical Ordering of Sequential Processes*", Acta Informatica, Vol 1, 1971, Pages 115-138.

[Dijkstra75] E. W. Dijkstra, "*Guarded Commands, Nondeterminacy and Formal Derivation of Programs*", CACM, Vol 18, No 8, August 1975, Pages 453-457.

[Dulay87] Naranker Dulay, Jeff Kramer, Jeff Magee, Morris Sloman and Kevin Twidle, "*Distributed System Construction: Experience with the Conic Toolkit*", In Experiences with Distributed Systems, Editor Jurgen Nehmer, Springer-Verlag, LNCS 309, 1987, Pages 189-212.

[Ericson82] Lars Warren Ericson, "*DPL-82: A Language for Distributed Processing*", In Proc. 3rd Intl. Conf. on Distributed Computer Systems, Miami/Fort Lauderdale, 18-22 October 1982, IEEE, Pages 526-531.

[Feldman79] S. I. Feldman, "*Make - A Computer Program for Maintaining Programs*", Software - Practice and Experience, Vol 9, April 1979, Pages 255-266.

[Foster86] David G. Foster, "*Separate Compilation in a Modula 2 Compiler*", Software - Practice and Experience, Vol 16, No 2, February 1986, Pages 101-106.

[Goldberg83] A. Goldberg and D. Robson, "*Smalltalk 80: The Language and its Implementation*", Addison-Wesley, 1983.

[Guttag85] John V. Guttag, James J. Horning, and Jeannette M. Wing, "*The Larch Family of Specification Languages*", IEEE Software, Vol 2, No 5 September 1985, Pages 24-36.

[Harel88] David Harel, "*On Visual Formalisms*", CACM, Vol 31, No 5, May 1988, Pages 514-530.

[Hayes86] John P. Hayes, Trevor Mudge, Quentin F. Stout, Stephen Colley and John Palmer, "*A Microprocessor-based Hypercube Supercomputer*", IEEE Micro, Vol 6, No 5, October 1986, Pages 6-17.

[Hayes87] Roger Hayes and Richard D. Schlichting, "*Facilitating Mixed Language Programming in Distributed Systems*", Vol SE-13, No 12, December 1987, Pages 1254-1264.

[Hoare68] C. A. R. Hoare, "*Record Handling*", In Programming Languages, Editor F. Genuys, Academic Press, 1968, Pages 291-347.

[Hoare84] C. A. R. Hoare, "*Communicating Sequential Processes*", Prentice-Hall International, 1984.

[Hewitt85] Carl Hewitt, "*The Challenge of Open Systems*", Byte, April 1985, Pages 223-242.

[Hillis85] W. D. Hillis, "*The Connection Machine*", MIT Press, 1985.

[Horning73] J. J. Horning and B. Randell, "*Process Structuring*", Computer Surveys, Vol 5, No 1, March 1973, Pages 5-30.

[Hughes83] J. W. Hughes and M.S. Powell, "*DTL: A Language for the Design and Implementation of Concurrent Programs as Structured Networks*", Software - Practice and Experience, Vol 13, 1983, Pages 1099-1112.

[Hughes89] John Hughes, "*Why Functional Programming Matters*", The Computer Journal, Vol 32, No 2, April 1989, Pages 98-107.

[ISO85] "*Specification of Abstract Syntax Notation One (ASN.1)*", Draft International Standard ISO/DIS 8824, TC97, 6 June 1985.

[Kahn88] Kenneth M. Kahn and Mark S. Miller, "*Language Design and Open Systems*", In The Ecology of Computation, Editor B. A. Huberman, Elsevier Science Publishers B. V. (North-Holland), 1988.

[Kaplan88] Simon M. Kaplan and Gail E. Kaiser, "*Garp: Graph Abstractions for Concurrent Programming*", In 2nd European Symp. on Programming ESOP '88, Nancy, France, March 1988, Editor H. Ganzinger, Springer-Verlag, LNCS 300, Pages 191-205.

[Kramer83] Jeff Kramer, Jeff Magee, Morris Sloman, and Andrew Lister, "*CONIC: an integrated approach to distributed computer control systems*", IEE Proc., Vol 130, Part E, No 1, January 1983, Pages 1-10.

[Kramer85] Jeff Kramer and Jeff Magee, "*Dynamic Configuration of Distributed Systems*", IEEE Trans. on Soft. Eng., Vol SE-11, No 4, April 1985, Pages 424-436.

[Kramer88] Jeff Kramer and Jeff Magee, "*A Model for Change Management*", In Proc. Workshop on the Future Trends of Distributed Computing Systems in the 1990s, Hong Kong, 14-16 September 1988, IEEE, Pages 286-295.

[Kramer89] Jeff Kramer, Jeff Magee, Keng Ng, "*Graphical Configuration Programming*", IEEE Computer, Vol 22, No 10, Pages 53-65.

[Kuck77] D. J. Kuck, "*A Survey of Parallel Machine Organisation and Programming*", ACM Computing Surveys, Vol 9, Pages 29-59.

[Lampson83] Butler W. Lampson, "*Hints for Computer System Design*", ACM Operating Systems Review, Vol 17, No 5, Proc. 9th ACM Symp. on Operating Systems Principles, Bretton Woods, New Hampshire, 10-13 October 1983. Pages 33-48. Also in IEEE Software, Vol 1, January 1984, Pages 11-28.

[Larus88] J. R. Larus and P. N. Hilfinger, "*Restructuring Lisp Programs for Concurrent Execution*", In ACM SIGPLAN PPEALS 1988 Parallel Programming Experiences with Applications, Languages and Systems, New Haven, Connecticut, July 19-21, 1988, SIGPLAN Notices, Vol 23, No 9, September 1988, Pages 100-110.

[LeBlanc82] Richard J. LeBlanc and Arthur B. Maccabe, "*The Design of a Programming Language Based on Connectivity Networks*", In IEEE Proc. 3rd Intl. Conf. on Distributed Computing Systems, Florida, October 1982, Pages 532-541.

[LeBlanc85] Thomas J. Le Blanc and Stuart Friedberg, "*HPC: A Model of Structure and Change in Distributed Systems*", IEEE Trans. on Computers, Vol 34, No 12, December 1985, Pages 1114-1129.

[Lee86] I. Lee, N. Prywes and B. Szymanski, "*Partitioning of Massive/Real-Time Programs for Parallel Processing*", In Advances in Computers, Vol 25, Editor M. C. Yovits, Academic Press, 1986, Pages 215-275.

[Leffler89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quaterman, "*The Design and Implementation of the 4.3BSD UNIX Operating System*", Addison-Wesley, 1989.

[LeLann81] Gerard LeLann, *"Motivations, Objectives and Characterization of Distributed Systems"*, In Distributed Systems - Architecture and Implementation, Editors B. W. Lampson, M. Paul and H. J. Siegert, Springer-Verlag, 1981, Pages 1-9.

[Lesser79] V. Lesser, D. Serrain and J. Bonar, *"PCL - A Process Oriented Job Control Language"*, In Proc. 1st Intl. Conf. on Distributed Computer Systems, Pages 315-329.

[Levy84] Michael R. Levy, *"Type Checking, Separate Compilation adn Reusability"*, In Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices, Vol 19, No 6, June 84, Pages 285-289.

[Lim82] Willie Y-P. Lim, *"HISDL: A Structure Description Language"*, CACM Vol 25, No 11, November 1982, Pages 823-830.

[Liskov83] Barbara Liskov and Robert Scheifler, *"Guardians and Actions: Linguistic support for robust, distributed programs"*, ACM TOPLAS, Vol 5, No 3, July 1983, Pages 381-404.

[Liskov84] Barbara Liskov, *"Overview of the Argus Language and System"*, Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, Februrary 1984.

[MacQueen86] David MacQueen, *"Using Dependent Types to Express Modular Structure"*, In 13th Annual ACM Principles of Programming Languages Symposium, St. Petersburg, January 1986, Pages 277-286.

[Magee84] Jeffrey N. Magee, *"Provision of Flexibility in Distributed Systems"*, PhD, Department of Computing, Imperial College, University of London.

[Magee86] Jeff Magee, Jeff Kramer and Morris Sloman, *"The Conic Support Environment forDistributed Systems"*, In Proc. of the NATO Advanced Study Institute on Distributed Operating Systems : Theory and Practice, Altinyunus, Cesme, Turkey 18-29, 1986. Published as Distributed Operating Systems, Editors Yakup Paker, Jean-Pierre Banatre, Muslim Bozyigit, NATO ASI Series, Vol F28, Springer Verlag, 1987, Pages 289-310.

[Milner80] Robin Milner, *"A Calculus of Communicating Systems"*, Springer-Verlag, LNCS 92, 1980.

[Minsky83] Naftaly H. Minsky, "*Localility in Software Systems*", In Proc. 10th ACM Symp. on POPL, Austin, Texas, January 24-26, 1983. Pages 299-312.

[Mitchell79] James G. Mitchell, William Maybury and Richard Sweet, "*Mesa Language Manual, Version 5.0*", Xerox Parc Rept. CSL-79-3 , Palo Alto Research Center, April 1979.

[Moor82] I. W. Moor, "*An Applicative Compiler for a Parallel Machine*", In Proc. ACM SIGPLAN 82 Symp. on Compiler Construction, Boston, SIGPLAN Notices, Vol 17, No 6, June 1982, Pages 284-293.

[Muhlenbein88] H. Muhlenbein, Th. Scheider, and S. Streitz, "*Network Programming with MUPPET*", Journal of Parallel and Distributed Computing, Vol 5, 1988, Pages 641-653.

[Ossher83] Harold L. Ossher and Brian Reed, "*Fable: A Programming Language solution to IC process automation problems*", In Proc. SIGPLAN '83 Symp. on Prog. Lang. Issues in Software Systems, San Francisco, June 27-29, 1983, SIGPLAN Notices, Vol 18, No 6, June 1983, Pages 137-148.

[Ossher87] Harold L. Ossher, "*A Mechanism for Specifying the Structure of Large, Layered Systems*", In Research Directions in Object-Oriented Programming, Editors: B. Shriver and P. Wegner, MIT Press, 1987, Pages 219-252.

[Parnas72a] David L. Parnas, "*A Technique for Software Module Specification with Examples*", CACM, Vol 15, No 5, May 1972, Pages 330-336.

[Parnas72b] David L. Parnas, "*On the Criteria for Decomposing Systems into Modules*", CACM, Vol 15, No 12, December 1972, Pages 1053-1058.

[Perry87] Dewayne E. Perry, "*Software Interconnection Models*", In Proc. 9th Intl. Conf. on Software Engineering, Monterey, California, March 1987, Pages 61-69.

[Quinn88] M. J. Quinn, P. J. Hatcher and K. C. Journdenais, "*Compiling C* Programs for a Hypercube Multicomputer*", In ACM SIGPLAN PPEALS 1988 Parallel Programming Experiences with Applications, Languages and Systems, New Haven, Connecticut, July 19-21, 1988, SIGPLAN Notices, Vol 23, No 9, September 1988, Pages 57-65.

[Randell86] B. Randell, "*System Design and Structuring*", The Computer Journal, Vol 29, Vol 4, August 1986, Pages 300-306.

[Ringwood88] G. A. Ringwood, "*Parlog86 and the Dining Logicians*", CACM, Vol 31, No 1, January 1988, Pages 10-25.

[Robbins84] David C. Robbins, "*Engineering a High-Capacity Pascal Compiler for High Performance*", In Proc. ACM SIGPLAN '84 Symp. on Compiler Construction, SIGPLAN Notices, Vol 19, No 6, June 84, Pages 300-309.

[Rudmik82] A. Rudmik and B. G. Moore, "*An Efficient Separate Compilation Strategy for Very Large Programs*", In Proc. ACM SIGPLAN 82 Symp. on Compiler Construction, Boston, SIGPLAN Notices, Vol 17, No 6, June 1982, Pages 301-307.

[Scheifler86] R. W. Scheifler and J. Gettys, "*The X Window System*", ACM Trans. on Graphics, Vol 5, No 2, April 1986, Pages 79-109.

[Schwanke88] Robert W. Schwanke and Gail E. Kaiser, "*Smarter Recompilation*", ACM TOPLAS, Vol 10, No 4, October 1988, Pages 627-632.

[Scott88] Michael L. Scott and Raphael A. Finkel, "*A Simple Mechanism for Type Security Across Compilation Units*", IEEE Trans. on Soft. Eng., Vol 14, No 8, August 1988, Pages 1238-1239.

[Seitz85] C. L. Sietz, "*The Cosmic Cube*", CACM, Vol 28, No 1, January 1985, Pages 22-33.

[Shapiro84] Ehud Shapiro, "*Systolic Programming: A Paradigm of Parallel Processing*", In Proc. of the Intl. Conf. on Fifth Generation Computer Systems 1984, edited by ICOT, November 1984, Pages 458-470.

[Sloman87] Morris Sloman and Jeff Kramer, "*Distributed Systems and Computer Networks*", Prentice-Hall International, 1987.

[Stone71] H. S. Stone, "*Parallel Processing with the Perfect Shuffle*", IEEE Trans. on Computers, Vol C-20, No 2, February 1971, Pages 153-161.

[Stefik86] Mark Stefik and Daniel G. Bobrow, "*Object-Oriented Programming: Themes and Variations*", The AI Magazine, January 1986, Pages 40-62.

[Strom83] Robert E. Strom and Shaula Yemini, "*NIL: An Integrated Language and System for Distributed Programming*", In Proc. SIGPLAN '83 Symp. on Prog. Lang. Issues in Software Systems, San Francisco, June 27-29, 1983, SIGPLAN Notices, Vol 18, No 6, June 1983, Pages 73-82.

[Stroustrup86] Bjarne Stroustrup, "*An Overview of* C++", ACM SIGPLAN Notices, Vol 21, No 10, October 1986, Pages 7-18.

[Sweet85] Richard W. Sweet, "*The Mesa Programming Environment*", In ACM SIGPLAN 85 Symp. on Lang. Issues in Prog. Environments, Seattle, Washington, 25-28 June 1985, SIGPLAN Notices, Vol 20, No 7, July 1985, Pages 216-229.

[Thakkar88] Shreekant Thakkar, Paul Gifford and Gary Fielland, "*The Balance Multiprocessor System*", IEEE Micro, Vol 8, No 1, February 1988, Pages 57-69.

[Thorelli85] Lars-Erik Thorelli, "*A Language for Linking Modules into Systems*", BIT, Vol 25, 1985, Pages 358-378.

[Tichy79] W. F. Tichy, "*Software Development based on Module Interconnection*", In Proc. 4th Intl. Conf. on Software Engineering, Munich, September 1979, Pages 29-49.

[Tichy86] W. F. Tichy, "*Smart Compilation*", ACM TOPLAS, Vol 8, No 3, July 1986.

[Turner86] David Turner, "*An Overview of Miranda*", ACM SIGPLAN Notices, Vol 21, No 12, December 1986, Pages 158-166.

[Turner87] David Turner, "*Functional Programming and Communicating Processes*", In Proc. of the Conf. on Parallel Architectures and Languages Europe (PARLE), Amstersdam, The Netherlands, June 15-19 1987, Editors J. W. de Bakker, A. J. Nijman and P. C. Treleaven, LNCS 259, Springer-Verlag, Pages 54-74.

[Walden84] K. Walden, "*Automatic Generation of Make Dependencies*", Software - Practice and Experience, Vol 14, No 6, June 1984, Pages 575-585.

[Watson87] P. Watson and I. Watson, "*Evaluating Functional Programs on the FLAGSHIP machine*", In Proc. 1987 Conf. on Functional Programming Languages and Computer Achitecture, Portland, Oregon, 1987, Pages 80-97.

[Weide82] Bruce W. Weide, Mark E. Brown, Jose A. S. Alegria, and Glen Meyer, "*A Graphical Interconnection Language and its Application to Concurrent Programming*", In Proc. 20th Annual Allerton Conference on Communication, Control and Computers, University of Illinois, October 1982.

[Wirth82] Niklaus Wirth, "*Programming in Modula-2*", Springer-Verlag, 1982.

[Wirth88a] Niklaus Wirth, "*Type Extensions*", ACM TOPLAS, Vol 10, No 2, April 1988, Pages 200-214.

[Wirth88b] Niklaus Wirth, "*The Programming Language Oberon*", Software - Practice and Experience, Vol 18, No 7, July 1988, Pages 671-690.

[Wolf85] Alexander L. Wolf, Lori A. Clarke and Jack C. Wileden, "*Ada-Based Support for Programming-in-the-Large*", IEEE Software, Vol 2, No 2, March 1985, Pages 58-71.

[Wybrabietz85] Dieter Wybrabietz and Rolf Massar, "*An Overview of LADY - A Language for the Implementation of Distributed Operating Systems*", SFB 124, Report No 12/85, Universitat Kaiserlautern, Fachbereich Informatik, 1985.

Appendix I Conic Configuration Language Definition

Version 3.0

N. Dulay
J. Kramer
J. Magee
M. Sloman
K. Twidle

July 1988

Department of Computing
Imperial College
University of London
180 Queens Gate
London SW7 2BZ

ABSTRACT

Conic provides an integrated approach to the design, implementation and management of **distributed** computer systems. The software methodology distinguishes between the **programming** of individual software modules and the **configuration** of a system from a set of instances of such modules. This distinction facilitates the programming of task modules without knowledge of the configuration in which they will be used, and also allows modification of a configuration without recompilation of its constituent modules.

This report defines the configuration language for Conic in which sequential task modules are identified, instantiated and interconnected into group modules. The configuration language also allows group modules to be identified, instantiated and interconnected allowing distributed systems to be built by hierarchic composition of task modules and group modules.

Keywords

Distributed programming; configuration language; message passing; module; port; task; interprocess communication.

Acknowledgements

Acknowledgement is made to the National Coal Board for a grant in aid of these studies, but the views expressed are those of the authors and not necessarily those of the Board.

Contents

1. Introduction

This report defines the Conic **configuration** language for building distributed systems. This is the language used to specify module instances and their interconnections [3]. The language used to program task modules is defined in a companion report [1].

1.1. Configuration Language Overview

Systems in Conic consist of interconnected sets of module instances, described by a configuration specification. Systems may be implemented as distributed or non-distributed configurations on single or multiprocessors. The configuration specification identifies the module types from which the system will be constructed, declares the instances of these types which will exist in the system and describes the interconnection of the instances by the links between their exitports and entryports. These three functions are termed **context definition**, **module instantiation** and **module interconnection** respectively.

The module types used in a configuration specification may be task modules containing a single sequential task or collections of modules called **group modules**. In Conic, group modules are configuration specifications and so define a module type that can be used in other configuration specifications. This allows systems to be constructed by the hierarchic composition of primitive task modules and composite group modules. Group modules that are distributable are termed **nodes**. The components of a node may share procedure and function code and pass pointer values in messages.

1.2. Programming Language Overview

The Conic programming language [1] is defined as an extension to ISO Pascal [4]. The unit of programming is the **task module**. The task module interface is specified by declaring typed entryports and exitports, and by declaring task module parameters. The other main extension to Pascal is the inclusion of message communication primitives.

The primitive operations on ports are **sending** and **receiveing** messages. The primitive operations support two types of message transaction: **request-reply** and **notify**. The request-reply transaction provides bidirectional, synchronous message passing, while the notify transaction is unidirectional and asynchronous. A **fail** clause may be used to withdraw from a request-reply send. There is, in addition, a **select** statement provided for selection from one of a set of ports from which messages may be received. Within a select, a **guard** can be used to mark a receive statement as ineligible for selection. A **timeout** can be used to withdraw from the select. In the case of the request-reply transaction, two further operations are provided: **forward**, in order to pass a request on to a third party for service, and **abort**, to cause the current transaction to fail.

In addition to the task module unit, a **definition module** unit is provided enabling tasks to be constructed in modular way.

2. Notation and Vocabulary

2.1. Syntax

The syntax is given as in the extended form of traditional BNF used in the Pascal Standard [4] and the Conic Programming Language Report [1], except that the metasymbol **xyz** (shown bolded) is used as an alternative to the metasymbol "xyz" to represent terminal symbols. The metasymbol [construct]* -- for zero or more repetitions is used rather than the metasymbol {construct}. UPPERCASE letters are not significant in non-terminal meta-identifiers, but serve as additional comments.

2.2. Special Symbols and Word Symbols

The Conic configuration language vocabulary consists of special symbols, identifiers, numbers and strings.

```
special-symbol =
        "[" | "]" | "(" | ")" | ":" | ";" | "," | "." | ".."
        | "#" | "'" | "+" | "–" | "*" | "/" | "=" | "<>"
        | "<" | ">" | "<=" | ">=" | "_" | word-
        symbol .
word-symbol =
        and | at | const | create | div | end |
        entryport | exitport | family | from |
        group | link | mod | module | not | or |
        reply | to | use | when .
```

2.3. Comments

The constructs

```
{     any sequence of characters not containing
      a right brace "}"
or
--    any sequence of characters not containing
      a new line
```

may be inserted between any two identifiers, constants and special symbols as a comment. Comments may be nested, and do not affect the meaning of the specification. The delimiters (* and *) may be used for the delimiters { and } respectively.

2.4. Identifiers

Identifiers are used as names of constants, datatypes, ports, definition modules, module types, module instances and ranges. Their association must be unique within their scope of validity, i.e. within the scope in which they are defined. The scope of a constant, datatype, port, definition module, module type or module instance identifier extends from its defining point (the point at which it is first introduced) to the end of the configuration specification. The scope of a range identifier extends from its defining point to the end of the construct in which it is defined.

```
id =
        identifier .
identifier =
        [ letter | break-char ]
        [ letter | digit | break-char ]* .
letter =
        "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
        "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
        "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
        "y" | "z" .
digit =
        "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
        "8" | "9" .
break-char =
        "_" .
```

Identifiers may be of any length. All characters (except embedded break-char's) are significant in distinguishing between identifiers. No identifier may have the same spelling as a word-symbol. The case (upper or lower) of letters is not significant.

The following identifiers are pre-defined:

constants	false, true, maxint, maxnat, maxlongint and signalvalue.
datatypes	boolean, char, integer, real, byte, natural, longint, address, signaltype, and string.
ports	implementation-defined.

2.5. Expressions

Expressions are a subset of the more general expressions found in Pascal and the Conic programming language. The main restriction is that variables, set expressions, and the value nil are prohibited, and that functions can only take value parameters. Range identifiers are allowable factors. Expressions are evaluated as in Pascal and the Conic programming language.

```
expr =
        expression .
expression =
        simple-expression [ relational-operator
        simple-expression ] .
simple-expression =
        [ sign ] term [ adding-operator term ]* .
term =
        factor [ multiplying-operator factor ]* .
factor =
        not factor | "(" expr ")" | CONSTANT-id |
        RANGE-id | number | character-string |
        string-string | function-designator .
relational-operator =
        "=" | "<>" | "<" | ">" | "<=" | ">=" .
adding-operator =
        "+" | "-" | or .
multiplying-operator =
        "*" | "/" | div | mod | and .
sign =
        "+" | "-" .
number =
        unsigned-integer | unsigned-real .
unsigned-integer =
        digit [ digit ]* [ "#" type-suffix ] |
        binary-digit [ binary-digit ]* "#2"
            [ type-suffix ] |
        octal-digit [ octal-digit ]* "#8"
            [ type-suffix ] |
        digit [ hex-digit ]* "#16"
            [ type-suffix ] .
type-suffix =
        "c" | "n" | "l" .
binary-digit =
        "0" | "1" .
octal-digit =
        "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .
hex-digit =
        digit | "a" | "b" | "c" | "d" | "e" | "f" .
```

```
unsigned-real =
        digits "." digits [ "e" [ sign ] digits ] |
        digits "e" [ sign ] digits .
character-string =
        " ’ " string-element [ string-element ]* " ’
        "
string-element =
        any-character-except-single-quote | " ’’’’ "
        .
string-string =
        " " " string-char [ string-char ]* " " " .
string-char =
        any-character-except-double-quote | " "" "
        .
```

3. Group Modules

A configuration specification defines a **group module** type (possibly with formal parameters) from which group module instances (with actual parameters) can be created. The specification identifies the component module types (which may be task module types or group module types), declares instances of these types, and describes the interconnection of these instances by the links between their exitports and entryports. These three functions are termed context definition, module instantiation and module interconnection respectively. Like the task module, the group module may have a message passing interface consisting of typed entryports and exitports. These ports can be linked to the ports of constituent module instances with the link construct (section 3.5).

In this report the word module is used as generic term for the terms task module and group module.

```
configuration-specification =
        group-module-header
        [ specification-part ]*
        end "." .
specification-part =
        constant-definition | context-definition |
        port-declaration | module-instantiation |
        module-interconnection .
```

3.1. Group Module Heading

The group module header specifies the name of the group module type and the formal parameters of the group module if any.

```
group-module-header =
        group module GROUP-MODULE-id
            [ "(" formal-parameters ")" ] ";" .
formal-parameters =
        parameter-specification
            [ ";" parameter-specification ]* .
parameter-specification
        constant-parameters | instance-parameters .
constant-parameters =
        CONSTANT-id [ "," CONSTANT-id ]*
        ":"
            STANDARD-DATATYPE-id
            [ default-value ].
default-value =
        "=" constant .
instance-parameters =
        MODULE-INSTANCE-id
            [ "," MODULE-INSTANCE-id ]*
        ":" MODULE-TYPE-id .
```

The formal parameters of a group module can be constant parameters or instance parameters. Constant parameters specify a constant identifier and its datatype. Instance parameters specify a module instance identifier and its module type, which must be imported into the group module via a context definition (section 3.3.1).

The value of a parameter is set by the corresponding actual parameter when the group module is instantiated.

The standard datatype identifiers boolean, char, integer, real, byte, natural, longint, address, signaltype and string are provided for all group modules. Their definitions are the same as those in the Conic programming language [1] and Pascal [4].

Default values can be specified for constant group module parameters. Group modules with default values need not have the corresponding actual parameters supplied.

<u>Examples</u>

group module bank (name:string="Midland"; sortcode:longint);

group module employee (name:string; branch:bank);

group module line (x,y:real=0.0);

3.2. Constant Definitions

As in Pascal, constant definitions can be used to introduce identifiers that denote specific constant values.

```
constant-definition =
        const constant-definition-part ";"
                [ constant-definition-part ";" ]* .
constant-definition-part =
        CONSTANT-id "=" constant .
constant =
        [ sign ] ( number | CONSTANT-id ) |
        character-string | string-string .
```

Examples

```
const
   max_ports=45;
   zero=0.0;
   day='monday';
   month="july";
```

3.3. Context Definitions (The Use Construct)

The types and constants from which the group module is constructed need to be imported into the group module by one or more context definitions.

```
context-definition =
        [ from-clause ]
        use context [ ";" context ]* ";" .
from-clause =
        from character-string .
context =
        module-type-context | datatype-context .
```

The from-clause specifies an implementation-dependent environment (e.g. a file, directory, pathname, database etc.) from which the specified definition modules can be accessed. If the from-clause is omitted, some default implementation-defined environment is assumed.

3.3.1. Importing Module Types

The module types from which the group module is constructed need to be imported into the group module by one or more module type contexts.

```
module-type-context =
        MODULE-TYPE-id .
```

Module type identifiers correspond to the identifiers used to name the module in the corresponding task or group module header.

Examples

```
from '/usr/lib/windows' use
   window, menu_manager, cursor;

from '../msdos' use
   dos_filesys;

use
   lance_driver;
```

3.3.2. Importing Constants, Datatypes and Functions

Common constants, datatypes, and functions can be imported from **definition modules** [1] using a datatype-context, which make them known inside the module.

```
datatype-context =
        DEFINITION-MODULE-id ":"
                object [ "," object ]* .
object =
        LOCAL-id [ "<" EXTERNAL-id ">" ] .
```

Definition-module-id specifies the name of the definition module from which the specified definitions are to be imported. The name that the definition, is to be known as, within the importing module is specified by local-id. External-id specifies the name that the definition is known as, within the specified definition module, if omitted, external-id defaults to local-id.

Note: Imported functions must not access variables global to the function, and only value parameters are permitted in imported functions.

Examples

```
from '/usr/lib/sys:/usr/fred/mydir' use
   object_defs : object_size, object_type;

use
   ascii : newline <lf>, return <cr>, stx;
```

3.4. Group Module Interface (Port Declarations)

Like task modules, group modules may have message passing interfaces that are specified by port declarations which specify a port name and its port type. Ports can be connected to the ports of instantiated modules with the link construct (section 3.6).

```
port-declaration =
        ( entryport | exitport )
        port-declaration-part
                [ ";" port-declaration-part ]* ";" .
port-declaration-part =
        port-list ":" REQUEST-MSG-TYPE-id
                [ reply REPLY-MSG-TYPE-id ] .
port-list =
        port-declarer [ "," port-declarer ]* .
port-declarer =
        PORT-id [ range ] .
range =
        "[" expr ".." expr "]" .
```

Message type may be any standard type or any imported datatype. Ports are declared with a port type which defines the type of value which may be sent or received (the port request type), and for request-reply transactions, the type of value which may be a reply (the port reply type). Notify entryports and notify exitports have no **reply** parts.

Families of entryports and exitports can be declared by suffixing a range with the port identifier. This is analogous to declaring arrays in Pascal. Ranges are restricted to being subranges.

<u>*Examples*</u>

```
exitport
   getch : char reply signaltype;
entryport
   openfile : open_req reply file_id;
exitport
   alarms [char] : boolean;
entryport
   std_channel [0..2] : channel_req reply integer;
```

3.5. Module Instantiation (The Create Construct)

Module instances are created from module types by one or more module instantiations. A module instantiation declares the name of the module instance, specifies the module type from which it is to be instantiated and optionally specifies a location for the created instance. If the module type has parameters, actual parameters may also need to be specified.

Families of module instances can be declared by defining a range identifier and an associated range, and using the range identifier to index family instances. The effect of a module family declaration is to repeat the create construct over the specified range with range-identifier taking successive values of the range. The scope of range-identifier is restricted to the create construct in which it is defined.

```
module-instantiation =
        [ when guard ]
        create [ family range-declarer ]
                [ at location ]
                instance-declaration
                [ ";" instance-declaration ]* [ ";" ]
guard =
        BOOLEAN-expression .
range-declarer =
        RANGE-id ":" range .
location =
        instance-name
instance-declaration =
        [ instance-declarer ":" ]
        MODULE-TYPE-id
                [ "(" actual-parameters ")" ] .
instance-declarer =
        MODULE-INSTANCE-id
                [ "[" RANGE-id "]" ] .
actual-parameters =
        actual-parameter [ "," actual-parameter ]*
        .
actual-parameter =
        positional-parameter | named-parameter |
        location .
positional-parameter =
        expression
```

```
named-parameter =
        CONSTANT-id "=" expression
instance-name =
        MODULE-INSTANCE-id [ "[" expr "]" ]
        .
```

Actual parameters must correspond in type to the formal parameters of the specified module type.

Actual parameters in create clauses can be specified by name (position-independent) or by position. Named parameters can be mixed with positional parameters. If the parameter following a named parameter is an un-named parameter, it is assumed to correspond to, the formal parameter following the formal parameter corresponding to the named actual parameter. Note: if an positional-parameter expression starts with an identifier that is the same name as any formal parameter of the module to be created, then a named parameter must be used, even if it is only of the form *parametername=parametername.*

If the specified module type has no default values, and no named parameters are used, then the actual parameters used must correspond in number, order and type to the formal parameters of the specified module type.

If a module instance name is not specified the module type identifier is overloaded and used as the module instance name. A module type identifier may only be overloaded once.

If a module instantiation begins with a when-clause, then the instance declarations following are only elaborated if the guard evaluates to true.

Instantiation of a module type causes the entryport and exitport names of the module type to be "inherited" by the module instance. The ports of an instantiated module are selected by prefixing the port name by the instantiated module name followed by a dot character (section 3.6).

Location specifies a module instance at which the created instances are to be co-located. If the location is omitted, the created instances will be located at the same location as the encapsulating group module instance i.e. the location is deferred until an instance of the group module type is created.

Examples

```
create family k : [1..max_alarms]
   alarm : window (0,k+(k-1)*15,k+k*15,12,18);

create
   interrupt : handler ( signal=SIGINT );

create family k : [1..elements] at transputer [k]
   fft [k]: fast_fourier;

create
   driver : serial_driver (retries=10);

when n>0 create
   next : myself (n=n-1);
```

3.6. Module Interconnection (The Link Construct)

Modules are connected together by linking source ports to sink ports. A source port is either a group module entryport (section 3.4) or a module instance exitport. A sink port is either a group module exitport or a module instance entryport. Linked ports must be of the same port type.

Families of modules and/or families of ports can be linked by defining one or more range identifiers and associated ranges, and using the range identifiers as constants in expressions that index the port or module families. The effect of a family linkage is to repeat the link construct over the specified ranges with range-identifiers taking successive values of their corresponding range. When more than one range is specified, repetitions are nested and performed in an analogous way to nested loops in Pascal; the first range being the outermost range, the last range being the innermost range. The scope of a range-identifier is restricted to the link construct in which it is defined.

```
module-interconnection =
        [ when guard ]
        link [ family-part ]
                link-specification
                [ ";" link-specification ]* ";" .
link-specification =
        source-port-list to sink-port-name |
        source-port-name to sink-port-list
source-port-list =
        source-port-name [ "," source-port-name
        ]* .
sink-port-list =
        sink-port-name  [ "," sink-port-name ]* .
source-port-name =
        ENTRYPORT-id |
        instance-name "." EXITPORT-id
                [ "[" expr "]" ] .
sink-port-name =
        EXITPORT-id  |
        instance-name "." ENTRYPORT-id
                [ "[" expr "]" ] .
family-part =
        family range-declarer
                [ "," range-declarer ]* .
```

Entryports may have more than one exitport linked to them. Notify exitports may be linked to more than one notify entryport. Request-reply exitports cannot be linked to more than one request-reply entryport.

If a module interconnection begins with a when-clause, then the links specifications following are only elaborated if the guard evaluates to true.

Examples

```
link
   multiplexor.transmit to line_driver.transmit;

link family k:[1..n]
   worker[k].transmit to line_driver.transmit;
   worker[k].received to line_driver.received;

link family k:[0..no_of_windows-1]
   alarm[k].out_string to window[k].window;
   window[k].write_out to console.write_string;
```

```
link family j:[0..ports-1]
    ring.xp [j] to ring.ep [(j+1) mod ports]

link family m:[1..mm], p:[1..pp]
    alpha[m].xport[p] to alpha[m].eport[p],
                         beta[m].eport[p],
                         gamma[m].eport;

    gamma[m].xport[p] to delta.eport[p],
                         group_xport[p];

    epsilon[m].xport[p],
    group_entryport to zeta[m].eport[p];

when (mm>4) and (pp>4)
link family m1:[1..mm], p1:[1..pp],
        m2:[1..mm], p2:[1..pp]
    psi[m1].xport[p1] to omega[m2].eport[p2];
```

References

[1] Dulay N., Kramer J., Magee J., Sloman M., Twidle K.: '*The Conic Programming Language - Version 3.0*', Conic Programmer's Manual, Dept. of Computing, Imperial College, July 1988.

[2] Sloman M., Kramer J.: '*Distributed Systems and Computer Networks*', Prentice-Hall International, 1987.

[3] Dulay N., Kramer J., Magee J., Sloman M., Twidle K.: '*Distributed System Construction: Experience with the Conic Toolkit*', Proc. Intl. Workshop on Experiences with Distributed Computer Systems, Kaiserslautern, Germany, September 1987, Publ. in Lecture Notes in Computer Science 309, Springer-Verlag, pp 189-212.

[4] ISO 7185: '*Specification for Computer Programming language Pascal*', International Standards Organisation. Also published as BS 6192:1982 by the British Standards Institute.

APPENDIX A EXAMPLES

Example 1 -- The Neo-Classic Patient Monitoring System

```
GROUP MODULE nurse (maxbed:integer);

   USE
         monmsg: bedtype, alarmstype;
   ENTRYPORT
         AlarmIn[1..maxbed] :alarmstype;
   EXITPORT
         BedOut[1..maxbed]  :signaltype REPLY bedtype;

   USE
            execcom; windman;

   CREATE  exec:execcom;
         ErrorW:windman(3,1,80,19,21);

   LINK
            exec.error TO ErrorW.window;

   { patient monitoring modules }

   USE
         disp; ncom; bedsel; alrm;

   CREATE
         display : disp(2);
         command : ncom;
         window : windman(0,15,65,1,11);
         selector : bedsel;

   LINK
         display.beddetails TO selector.BedIn;
         command.bedselect TO selector.bedselect;
         display.stdwrite TO window.window;


   CREATE  FAMILY k:[1..maxbed]
         AlarmD[k] : alrm(k);
         AlarmW[k] : windman(0,k+(k-1)*15,k+k*15,12,18);

   LINK    FAMILY k:[1..maxbed]
         AlarmD[k].std_write  TO  AlarmW[k].Window;
         AlarmIn[k]  TO  AlarmD[k].alarminput;
         selector.BedOut[k]  TO  BedOut[k];

END.
```

```
GROUP MODULE patient;

   USE
         monmsg: bedtype, alarmstype;
   EXITPORT
         alarmoutput : alarmstype;
   ENTRYPORT
         BedIn : signaltype REPLY bedtype;

   USE
         execcom; windman;

   CREATE
         exec:execcom;
         ErrorW:windman(6,1,80,13,15);

   LINK
         exec.error  TO  ErrorW.window;

   { patient monitoring modules }

   USE
         sim; monit; disp; com;

   CREATE
         scanner : sim(2);
         monitor : monit;
         display : disp(2);
         command : com;
         window  : windman(0,15,65,1,11);

   LINK
         scanner.sensoroutput TO  monitor.sensorinput;
         display.beddetails  TO  monitor.beddetails;
         command.newpatient  TO  monitor.change;
         display.stdwrite  TO  window.window;

         BedIn  TO  monitor.beddetails;
         monitor.AlarmOutput  TO  AlarmOutput;

END.
```

```
GROUP MODULE ward (nbed:integer);

   USE
         patient;
         nurse;
         transputer;

   CREATE  FAMILY  k:[1..nbed+1]
         node : transputer(k);

   CREATE  AT node[nbed+1]
         nurse (nbed);

   CREATE  FAMILY  k:[1..nbed] AT node[k]
         bed[k]:patient;

   LINK   FAMILY k:[1..nbed]
         nurse.BedOut[k]  TO  bed[k].BedIn;
         bed[k].alarmoutput  TO  nurse.AlarmIn[k];

END.
```

Example 2 -- Batcher's Bitonic Sort

```
#
GROUP MODULE batcher(n:integer=8);
#include <node.h>

USE     sort;
        interface;

CREATE
        sort(n=n);
        interface(n=n);

LINK FAMILY i:[0..n-1]
        interface.out[i]  TO  sort.input[i];
        sort.output[i]  TO  interface.inp[i];
END.
```

```
GROUP MODULE sort(n:integer);

ENTRYPORT input[0..n-1]:integer;
EXITPORT output[0..n-1]:integer;

USE    bitonic;
CREATE  bitonic(n=n);

LINK FAMILY i:[0..n-1]
        bitonic.output[i]  TO  output[i];

WHEN n>2 CREATE
        ascend:sort(n= n DIV 2);
        descend:sort(n= n DIV 2)

WHEN n>2 LINK FAMILY i:[0..(n DIV 2) -1]
        input[i]  TO  ascend.input[i];
        input[(n DIV 2) +i]  TO  descend.input[i];
        ascend.output[i]  TO  bitonic.input[i];
        descend.output[(n DIV 2)-1-i]  TO  bitonic.input[(n DIV 2)+i];

WHEN n=2 LINK FAMILY i:[0..1]
        input[i]  TO  bitonic.input[i];

END.
```

```
GROUP MODULE bitonic(n:integer);

ENTRYPORT input[0..n-1]:integer;
EXITPORT  output[0..n-1]:integer;

USE     comparator;

CREATE FAMILY i:[0..(n DIV 2)-1]
        ce[i]:comparator;

LINK FAMILY i:[0..(n DIV 2)-1]
        input[i]  TO  ce[i].a;
        input[(n DIV 2)+i]  TO  ce[i].b;

WHEN n>2 CREATE
        low:bitonic(n= n DIV 2);
        high:bitonic(n= n DIV 2);

WHEN n>2 LINK FAMILY i:[0..(n DIV 2)-1]
        ce[i].low  TO  low.input[i];
        ce[i].high  TO  high.input[i];
        low.output[i]  TO  output[i];
        high.output[i]  TO  output[(n DIV 2)+i];

WHEN n=2 LINK
        ce[0].low  TO  output[0];
        ce[0].high  TO  output[1];
END.
```

```
TASK MODULE comparator;
ENTRYPORT
        a,b:integer;
EXITPORT
        low,high:integer;
VAR
        av,bv:integer;
BEGIN
        LOOP
                RECEIVE av FROM a;
                RECEIVE bv FROM b;
                IF av<=bv THEN BEGIN
                        SEND av TO low;
                        SEND bv TO high;
                        END
                ELSE BEGIN
                        SEND bv TO low;
                        SEND av TO high;
                END;
        END;
END.
```

```
TASK MODULE interface(n:integer);
EXITPORT
        out[0..127]:integer;
ENTRYPORT
        inp[0..127]:integer;
VAR
        input,output:text;
        i,v:integer;
BEGIN
        LOOP
                write(n:1,'>');
                flush(output);
                FOR i:=0 TO n-1 DO
                BEGIN
                        read(v);
                        SEND v TO out[i];
                END;
                readln;
                write('sorted:- ');
                FOR i:=0 TO n-1 DO
                BEGIN
                        RECEIVE v FROM inp[i];
                        write(' ',v:1);
                END;
                writeln;
                flush(output);
        END;
END.
```

Appendix II Symbol File Syntax

Metalanguage

The format of symbol files is given in a boxed form of BNF as defined below:

Terminal symbols are written in UPPERCASE.

Non-terminals are written in lowercase with the first letter of each word in upper case.

Non-terminals may be prefixed by a label and colon (:). Labels are used purely for exposition.

A * suffix denotes zero or more repetitions.

A + suffix denotes one or more repetitions.

Productions are shown boxed, with the rule specified in the box and the non-terminal above the box.

```
            Symbol File
+---------------------------------+
|      Magic Number Section       |
|     Directory Table Section     |
|        Component Section        |
|       Parameters Section        |
|       Identifier Section        |
|     Type Extensions Section     |
|        Debugging Section        |
+---------------------------------+

        Magic Number Section
+---------------------------------+
|      Magic Number : Integer     |
+---------------------------------+

       Directory Table Section
+---------------------------------+
|         Length : Integer        |
|     Directory Name: String *    |
+---------------------------------+

          Component Section
+---------------------------------+
|     Comp : Component ID +       |
+---------------------------------+
```

Null

Component ID

Name : String
Comp Number : Integer
Home : Integer
Time Stamp

Time Stamp

Epoch : Longint
Unix Process Id : Integer

Parameters Section

Parameter Identifier
Null

Identifier Section

Identifier *
Null

Identifier

(Port Identifier | Type Identifier | Constant Identifier | Field Identifier | Parameter Identifier | Nil Identifier)

Common Id Part

Id Name : String
Owner : Integer
Id Type : Type Structure
Next : Identifier

Nil Identifier

Id Kind : Null

Port Identifier

ID Kind: PORT
Common Id Part
Port Kind: (EP | XP)
Port Number : Integer

Type Identifier

Id Kind : TYPE
Common Id Part

Constant Identifier

Id Kind : CONSTANT
Common Id Part
Value : (Integer | String)

Field Identifier

Id Kind : FIELD
Common Id Part
Offset : Integer

Parameter Identifier

Id Kind : PARAMETER
Common Id Part
Parameter Position : Integer

Type Structure

(Standard Type | Embedded Type | Read type | Message Type | Subrange Type | Enumerated Type | Array Type | Record Type | Set Type | Pointer Type)

Standard Type
Key : (NIL | BOOLEAN | CHAR | INT | REAL | BYTE | NATURAL | LONGINT | STRING)

Read Type
Key : Integer

Common Type Part
Key : Integer
Byte Size : Integer
Packed : Boolean

Message Type
Common Type Part
Form : MESSAGE
Request Type : Type Structure
Reply Type : Type Structure

Enumerated Type
Common Type Part
Form : ENUMERATED
First : Constant Identifier

Subrange Type
Common Type Part
Form : SUBRANGE
Low Value : Integer
High Value : Integer
Base Type : Type Structure

Array Type
Common Type Part
Form : ARRAY
Index Type : Type Structure
Element Type : Type Structure

Record Type
Common Type Part
Form : RECORD
Super Type : Type Structure
First : Field Identifier

Set Type
Common Type Part
Form : SET
Base Type : Type Structure

Pointer Type
Common Type Part
Form : POINTER
Base Type : Type Structure

Embedded Types
Zero Key : Integer
Type Identifier
New Key : Integer

Type Extension Section
Extension *
Null

Extension
Subtype Owner : Integer
Subtype Key : Integer
Supertype Owner : Integer
Supertype Key : Integer

String
Non-Zero Bytes : Byte *
Null

Longint
Low Integer : Integer
High Integer : Integer

Integer
Low Byte : Byte
High Byte : Byte

Null
Zero Value : Byte

Byte
Value in 0..255