∂arwin Overview

∂arwin is a language for describing hierarchic configuration structures. ∂arwin is termed a configuration language because it is intended to be purely used to describe and control the configuration structure of a program. It is thus a language for "programming-in-the-large". Unlike other module-interconnection languages, ∂arwin addresses the dynamic aspects of configuration as well as providing for static configuration.

Although normally used to describe logical (software) configurations, ∂arwin can also be used to describe physical (hardware) configurations. A scheme for mapping logical configurations to physical configurations has been explored (See documentation for **red**).

In this document, we outline the concepts and abstractions in ∂arwin. Diagrams are used to show the structures and reinforce the descriptions.

A ∂arwin configuration structure can be viewed as a hierarchy of interconnected **component instances**. Each level of the hierarchy being described with a separate ∂arwin configuration description termed a **composite component type**. The leaves of the hierarchy represent component instances without a ∂arwin sub-configuration. Typically these are instances of 'real' software modules, or represent 'real' pieces of hardware. Such instances are termed primitive and are instantiated by ∂arwin from **primitive component type**s. Primitive component types can be described with a subset of the ∂arwin language provided for composite components. The implementation and sub-structuring of primitive component types (*programming-in-the-small*) is left as the responsibility of individual software and hardware providers.

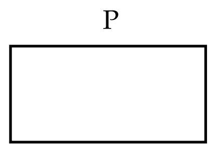
Primitive Components: Processes

The form of primitive component provided by the Regis C++ implementation is the process. The choice of such autonomous components is based on their suitability for concurrent, distributed and parallel programming. Other kinds of component (such as modules or procedures) may be more appropriate in other contexts. Parallelism within processes (*fine-grain parallelism*) is left for individual programming language compilers to identify and translate for given target machines.

Minimally primitive components must make available to the configuration language a component type name. For example, the component type:

> **component** P;

defines a component type with name P. This component can be represented graphically as follows:

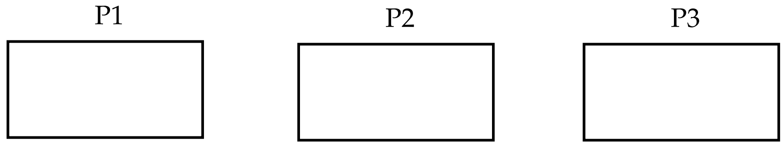


Graphical Representation of a Component Type

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

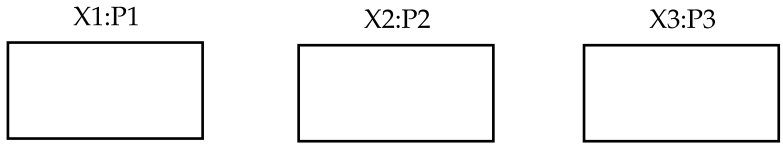
Given a set of named component types, P1, P2, and P3:

component P1; **component** P2; **component** P3;



we can declare a set of component instances X1, X2, X3, with the instance declaration:

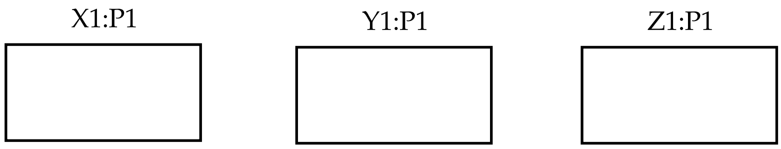
inst X1 : P1; X2 : P2; X3 : P3;



Instantiation of Components

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

inst X1 : P1; Y1 : P1; Z1 : P1;



Multiple Instantiation of a Component

Configuration declarations of this form are conceptually simple, clear, and declarative. They provide an easy to understand building-block approach to software construction. They do not, however, explicitly expose the underlying interaction structure of a program. The need to show and promote such structures is important as it allows components to be defined in a *configuration-independent* way. A configuration-independent component is one that is written without it needing to know which specific components it will interact with, but which has a specific interaction interface.

Another important consideration is the requirement in many systems of *interaction safety* , that is, to be able to restrict and prevent unintended interactions.

The *interaction structure* of a program can be specified by **binding** components together using ∂arwin. Approaches where the binding of components is embedded within the component programming language (termed the direct binding approach) are harder to comprehend although they are normally more powerful. The alternative to direct binding is indirect binding.

Indirect Binding

Two approaches to indirect binding have been explored, the global or *shared memory* approach and the *network* approach. 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 parallel environment can be highly inefficient compared to the 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 by binding all other processes to these processes it appears that the network model is more general. (We can argue the other way also!)

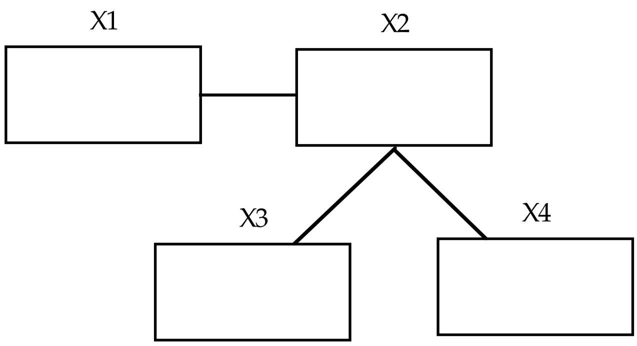
The network approach is a natural analogue of actual computer networks, and is adopted in ∂arwin. The network approach has also been modelled in functional programs [Turner87] and concurrent logic programs [Shapiro84].

By making interaction structure and interaction safety a configuration concern, and by making component binding indirect, leads to programs as networks of loosely-coupled, configuration-independent components (Lego approach to software).

Bindings

The interaction structure of programs could be specified by enumerating all possible component interactions. For example

```
bind  X1 – X2;
      X2 – X3;
      X2 – X4;
```



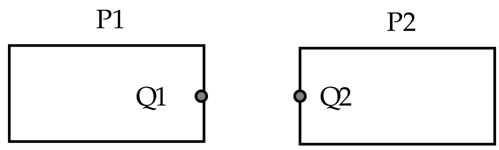
Binding of Components

This is a simple, yet powerful approach to structuring programs. Binding declarations 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 *provide*d by and/or *require*d by individual components must be made, and checks done to ensure that component requirements and component provisions are bound safely. Having such precise controls contributes to reductions in software errors.

Interfaces

In order to establish more precise controls, component types need to define a more elaborate interface. In ∂arwin an interface is a set of names. These interface names are implicitly instanstiated during component instantiation. For example: given a component P1 with an interface name Q1, and component P2 with an interface name Q2:

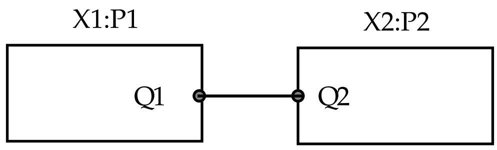
component P1{Q1}　　　　**component** P2 {Q2}



Interface Names

we can declare and bind instances X1 and X2 of these component types as follows:

inst X1 : P1; X2 : P2;
bind X1.Q1 – X2.Q2;



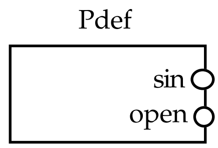
Binding of Interface Names

Directionality: Provides and Requires

Interface names are further refined in ∂arwin into **provide**d names, and **require**d names.

Required interface names identify services *required* by a component, for example, the ∂arwin component type Pdef:

component Pdef { **require** sin; open; }

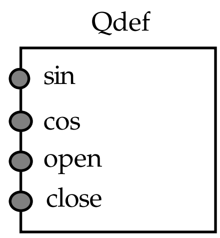


Required Services

specifies a component Pdef with requirements for services sin and open. Services normally represent some form of inter-component communication.

Provided interface names identify services *provided* by a component, for example, the ∂arwin component type Qdef:

component Qdef {**provide** sin; cos; open; close}



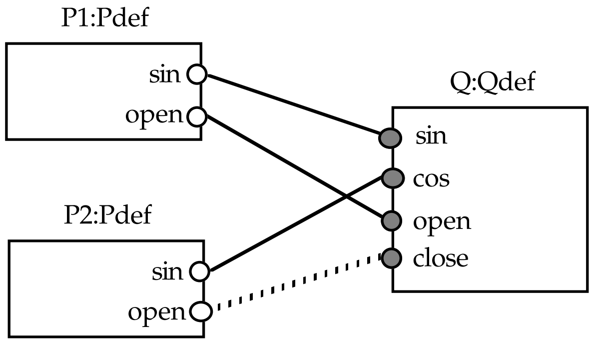
Provided Services

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

Safety interaction is assisted by only allowing bindings of interface names that have compatible directionality, i.e. if one end of the binding is an required name, then the other end must be a provided name. We show in later section the ∂arwin approach to typing interaction names and ensuring bindings are “type-safe”.

Example:

> **inst** P1 : Pdef; P2 : Pdef; Q : Qdef;
> **bind** P1.sin -- Q.sin
> **bind** P1.open -- Q.open
> **bind** P2.sin -- Q.cos
> **bind** P2.open -- Q.close



Bindings of Required names to Provided names

The binding of P2.sin to Q.cos is legal, since the safety rules have been met. The safety rule does not require names to be identical as in CCS.

The division of interfaces into required names and provided names [Tichy79, Wolf85] is crucial to ensuring configuration-independent components. In languages without this division, the binding between components is normally determined by explicit imports made in the specification parts of components. 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 components are direct, e.g component A imports and calls service R from component B where component B, is named explicitly within component 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 leads to components that are loosely coupled, configuration independent and re-configurable.

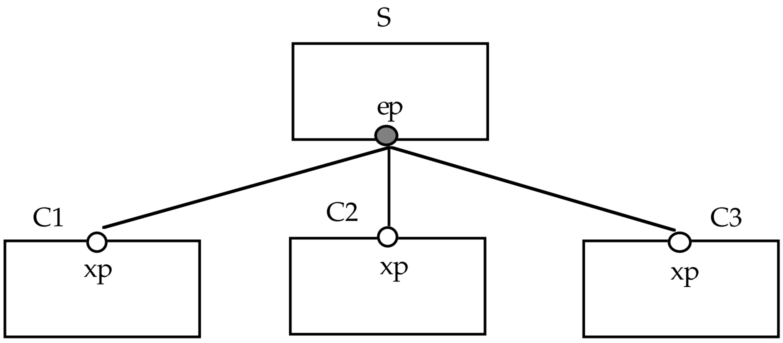
In systems such as [Minsky83, Wolf85], components can name not only the items to export, but also which components can import them. This gives the component implementor greater control on the usage of components but also leads to inflexible tightly coupled components, that are less re-usable.

A more deductive approach is taken in [Levy84] where the requirements of a component are automatically determined by the programming language compiler in terms of the functions called but not defined within the component. 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. For example problems can occur if there are functions whose names and parameters clash.

Ideally it would be desirable to attach the behaviour specification of services to interaction points, 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.

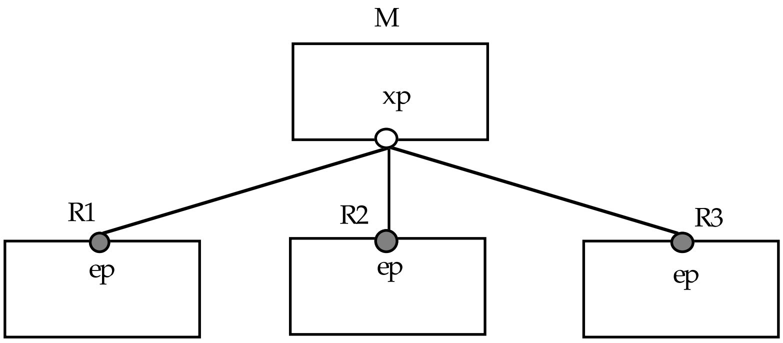
Fan-In and Fan-Out

The binding rules of ∂arwin do not preclude the binding of provided names to more than one required name, or required names to more than one provided name. Many-to-one bindings of the form:



Fan-In

are allowed. One-to-many bindings of the form:



Fan-Out

are not currently supported in ∂arwin. A future version of ∂arwin will support the notion of an interface name that identifies a set of services, and will allow One-to-Many bindings. Fan-in is useful for specifying server-client interactions, while fan-out is useful for multi-cast interactions. Of course the interaction behaviour of programs with fan-in and/or fan-out is dependent on the programming language of primitive components.

Composite Components

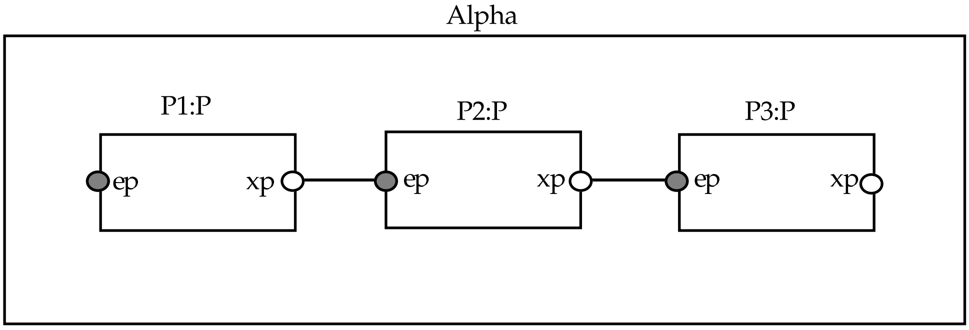
Composite components unify the concepts presented so far into a coherent and flexible structuring abstraction. Composite components are the means to specifying large structures in a modular way.

Encapsulation

Composite components provide an encapsulation mechanism for program structures. The following composite component type encapsulates the three sub-component instances P1, P2 and P3 of component type P:

```
component P {provide ep; require xp;j}

component Alpha {
        inst P1 : P;  P2 : P;  P3 : P;
        inst P1.xp -- P2.ep;
        inst P2.xp -- P3.ep;
```

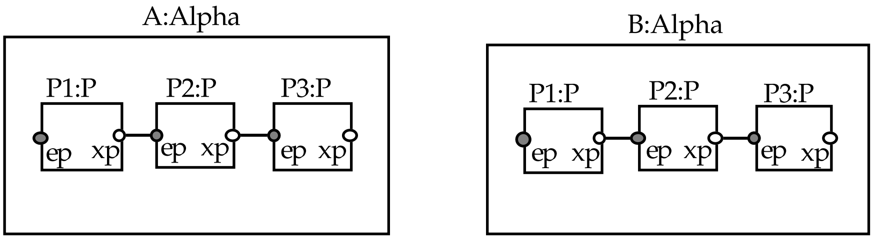
}



Example of a Composite Component

Composite component types can be substituted for primitive component types, instance declarations. Like primitive component types, composite component types can be multiply-instantiated, as in the following

inst A : Alpha; B : Alpha;



Multiple Instantiation of a Composite Component

The uniform treatment of composite and primitive components, leads to a black-box approach to components. It is not possible to distinguish between components that are primitive and components that are composite. Primitive components can be replaced by composite components and vice-versa.

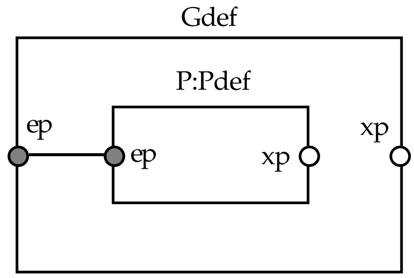
Since composite components can be instantiated in other composite components, composite components support hierarchic program structuring better than modular programming languages which disallow modules of modules or multiple instantiation of a single module.

Hierarchic Binding

Composite components also support interface declarations. This is achieved with the same mechanism used for primitive components, namely component interfaces. Only the name of the component, and its interface names are visible outside the component type definition. The correspondence between interface names and sub-instances declared within a composite component is specified by allowing hierarchic bindings. There are three types of hierarchic binding supported in ∂arwin:

Inbound Hierarchic Binding

bind providedName **–** Instance.providedName

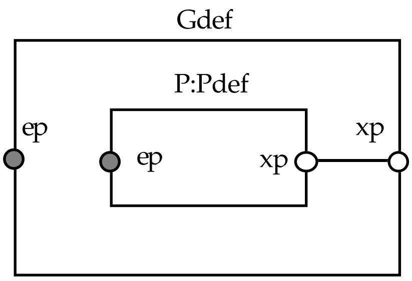


Inbound Hierarchic Binding

This form of hierarchic binding exports" out a provision of a sub-instance.

Outbound Hierarchic Binding

bind Instance.requiredName **to** requiredName

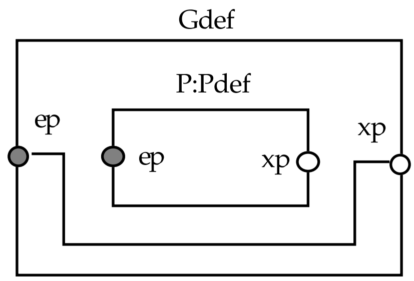


Outbound Hierarchic Binding

This form of hierarchic binding exports out a requirement of a sub-instance.

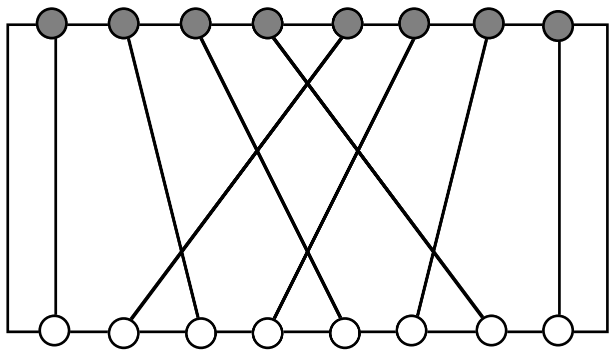
Forwarded Binding

bind providedName **to** requiredName



Forwarded Hierarchic Binding

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

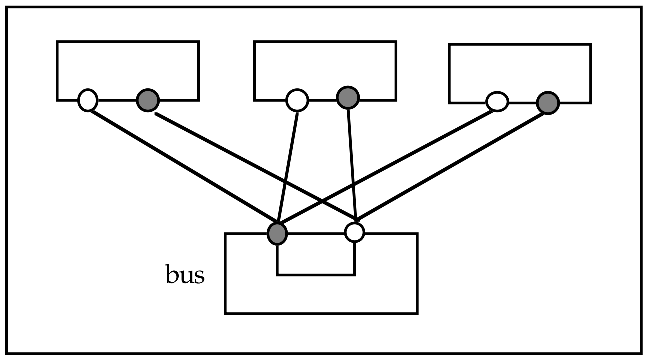


Example of a Perfect Shuffle Switching Structure

is interconnected as a perfect shuffle [Stone71].

Logical Bus

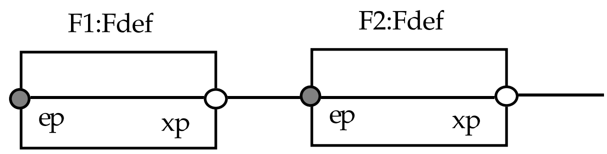
Another use for forwarded binding is in the declaration of local logical buses. These comprise a single composite component with a forwarded binding as in:



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. Note: logical buses cannot be written in the current form of ∂arwin as One-to-Many bindings are not supported. Forwarded binding also displays the transitive property

Transitive Bindings



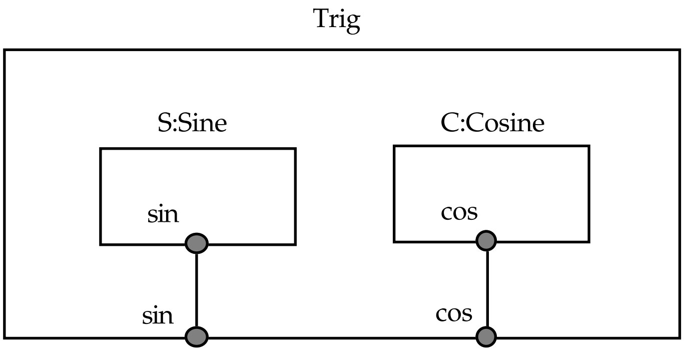
Transitive Bindings

Incremental Structuring

Incremental structuring allows new components to be constructed from old components, new services to be added or existing services to be changed. An alternative way of reducing incremental plumbing is described in section (on Derived Structures)

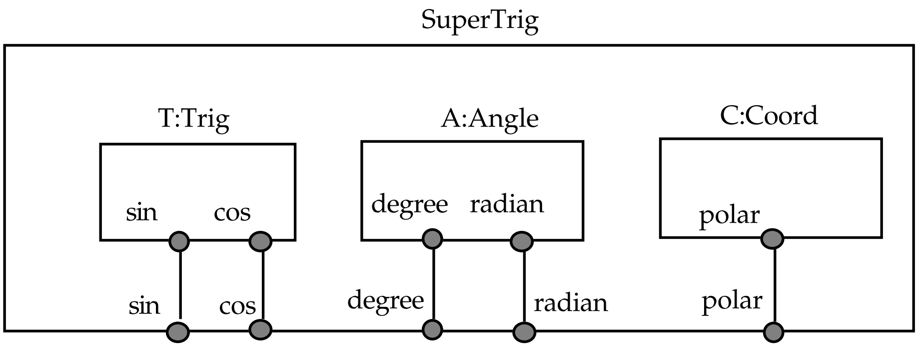
Composite components are sufficiently powerful to construct components by defining the differences. The following examples illustrate this.

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



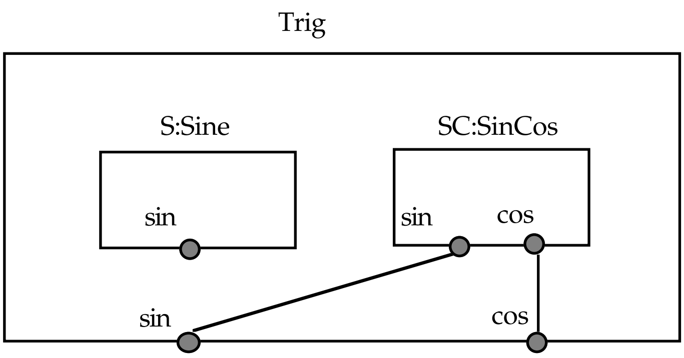
Incremental Structuring of Components

New components can be constructed from several existing components also:



Incremental Structuring from Several Components

Conflicts are handled by renaming, since name overloading is not currently supported. Component requirements can be incrementally exported also. A new component can be defined that overrides an existing component as in:



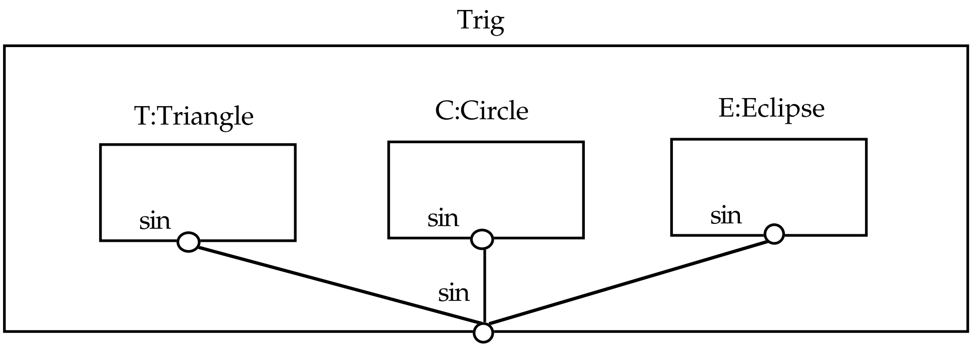
Overriding Components

Coalescing, Multicasting and Loop Backs

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

Hierarchic Coalescing (Multiplexing)

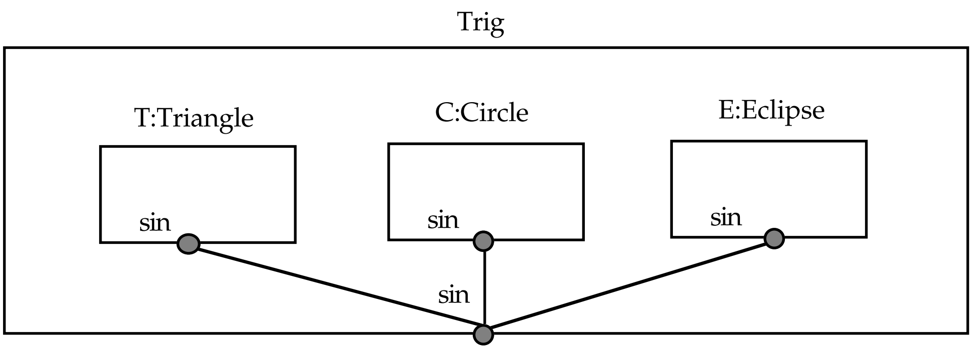
Several sub-component requirements can be bound to one requirement:



Hierarchic Coalescing

Hierarchic Multicasting

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

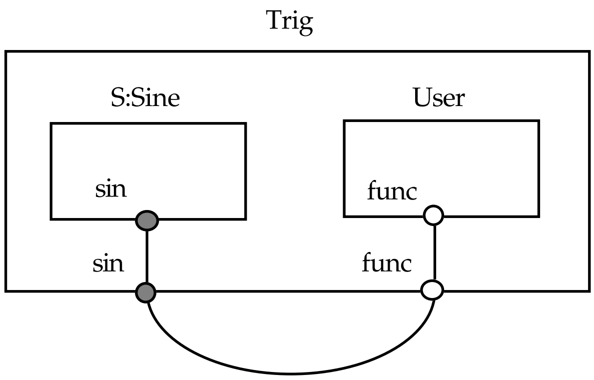


Hierarchic Multicasting

Hierarchic 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. Note: multi-casting is not currently supported as one-to-many bindings are not allowed.

Loop Backs

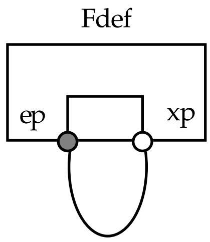
The rules for component binding allow provisions to be bound to requirements. This rule allows a requirement to be bound back to a provision of the same component. This allows the configuration programmer to either provide his own service, or use a service provided by the component, or ignore the requirement.



Loop Back Binding

A possibility exists for establishing circular binding, such as:

Circular Binding



Circular Binding

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

Advanced Features

Compositive components are a powerful mechanism for structuring programs in a modular way. In order to increase expressiveness, ∂arwin supports parameterisation, replication, variation, recursion, and dynamic elaboration. In order to increase re-usability, ∂arwin supports generic structures, and derived composite components (inheritance).

Parameterised Structures

Parameters are used to control the size and topology of ∂arwin structures and also to supply initial values to primitive components. Component parameter declarations are written in the style of C parameter declarations, for example:

component A (**int** a, **char** ch)

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

inst celsius : thermometer (45,'C')
inst fahrenheit : thermometer (93,'F')

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

component thermometer (**int** initial=98, **char** unit='F')

The basic scalar types currently supported by ∂arwin are **int**, **double**, **char**, **char** *. Expressions are a subset of those found in C.

Components as Parameters (Generic Structures)

As well as simple values, component types can also be passed paramterically, leading to generic structures:

Example:

component driver (**int** copies, **component** printer) {
    inst p : printer (copies);
}

component user (**int** copies) {
    inst d1 : (copies, postscript);
    inst d2 : (copies, quickdraw);
}

Replicated Structures

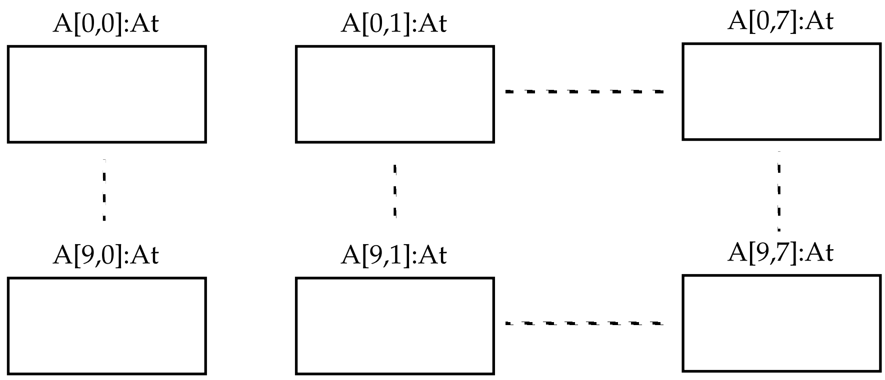
Replicated or array structures occur so frequently in practice that their inclusion in ∂arwin is almost obligatory.

Replication occurs in two places in ∂arwin, replicated component instances, and replicated interface names.

Replicated Instances (Instance Arrays)

Replicated instances require an **array** declaration prior to instanstiation to define the dimensions, bounds and component type of the instance arrays, e.g.:

array A [10][8] : Atype;



Note: Arrays in ∂arwin are like those in C and are indexed from element 0. In ∂arwin however, arrays bounds can be specifed with expressions that are evaluated at runtime, e.g. component parameters can be used in expressions to control the size and topology of replicated structures.

Instances of this type can be declared individually, e.g. the first element could be declared with:

inst A[0][0] (90, ‘F’);

More usually a **forall** declaration is used to iterate over elements of the array, e.g.

forall I : 0..9 **forall** K : 0..7
    inst A[I] [K] (I, ‘F’)

Note the passing of the forall identifier I as a parameter to individual replicated instances. Programs with quite complex interaction topologies can be expressed using foralls, e.g. pipes, rings, grids, hypercubes, butterflies, switching networks and tree topologies.

Replicated Interface Names (Interface Arrays)

Replicated requirements and provisions are declared by suffixing the name with the dimensions and bounds as in:

```
require req [10] [10];
provide prov [n*2];
```

Replicated Binding

Replicated instances and interfaces can selected by supplying an indexing expression in square brackets after the replicated instance/interface name. This expression can include component parameters as well as forall identifiers.

```
forall K : 0..N
        bind A[K].xp[K] -- req [N-K]
```

Variant Structures

It is often convenient to allow a component to have the potential to define more than one configuration structure, and to be able to defer the choice until the component is instantiated. This is achieved by allowing component instantiation and component binding declarations to be nested within a guarded (**when**) declaration:

```
when guard inst .....
when guard bind ...
```

The guard is a boolean expression that typically includes one or more component parameters which control the form of variant structure produced.

Guarded declarations can also be used to guard compounded declarations, e.g.:

```
when guard {
        inst ...
        bind..
        when guard2 {
                ....
        }
}
```

Recursive Structures

A structure is said to be recursive if it is defined in terms of itself. A characteristic feature of recursive structures is their ability to vary in size.

In ∂arwin, component types can be recursively instantiated within their own definition, for example:

```
component a {
        inst aa : a;
        inst  b;
}
```

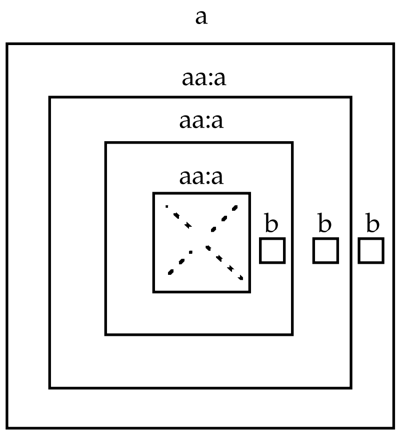


Fig 3.36 Recursive Structuring

This particular example defines an infinite program structure, 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:

```
component a (int k) {
        when k>0 {
                inst aa : a (k-1);
                inst b;
        }
}
```

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

Lazy Structures

In order to lazily elaborate a structure, instances can be declared as dynamic as in:

```
inst d : dyn comp;
```

Instances are normally created whenever the parent instance is created. Dynamic instances however are only created when a bound provision is actually "called"/"invoked"/"triggered". This gives ∂arwin the potential for lazy instantiation.

Example:

```
inst a: atype;
inst d: dyn dtype;
bind a.send -- d.wait;
```

In this example, d will be created the first time d.wait is "called".

Dynamic Structures

Example:

```
inst a: atype;
bind a.createB -- dyn B;
```

In this example, each call to a.createB will instantiate a new B along with any bindings from the new B.

Open Systems

Provided names can be exported to an external namespace by binding to a **export** name, whilst required can be imported by binding to an **import** name.

```
export  fred  @ "namespace";
bind fred -- mycomp.fred;
import jim @ "namespace";
bind mycomp.jim -- jim;
```

Derived Structures

As well as nested structuring, ∂arwin also supports incremental structuring by extension. This is achieved by allowing component types to be declared as derived component types (aka C++).

Example:

```
component alpha (int n) {
        require reqa;
        inst a : atype (n);
}

component beta (int n=10) : alpha (n*2) {
        require reqb;
        inst b : btype (n);
}

component gamma (int n) : beta, alpha (5) {
        require reqc;
        inst c : ctype (n);
        bind c.out -- b.in;
        bind b.out -- a.in;
}
```

In this example the instantiation of component gamma with parameter 3 say, causes the following elaboration:

```
gamma (3)
        beta (10)
                alpha (20)
                        inst a : atype (20)
                inst b : btype(10)
        inst c :ctype (3)
        bind c.out -- b.in;
        bind b.out -- a.in;
```

The interface of gamma is the union of all the interfaces of the base components e.g. {require reqa, reqb, reqc}

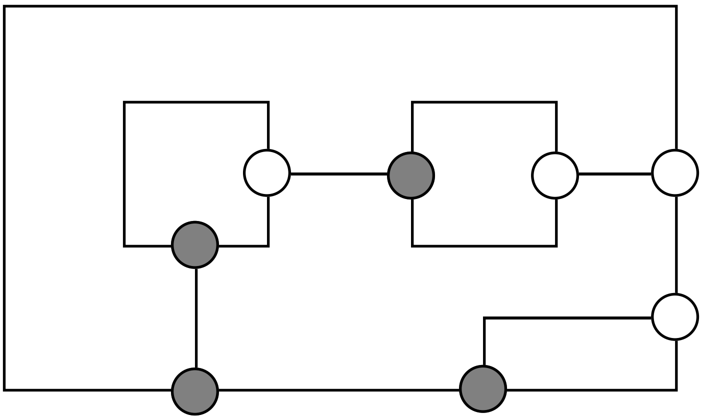
Note: if base component occurs more than once only the first reference to it has any effect.

Interaction Safety

In order to restrict and prevent unintended interactions, ∂arwin provides three kinds of constraint checks, directionality constraints, connectivity constraints and typing constraints. Having such precise controls contributes to reductions in software errors.

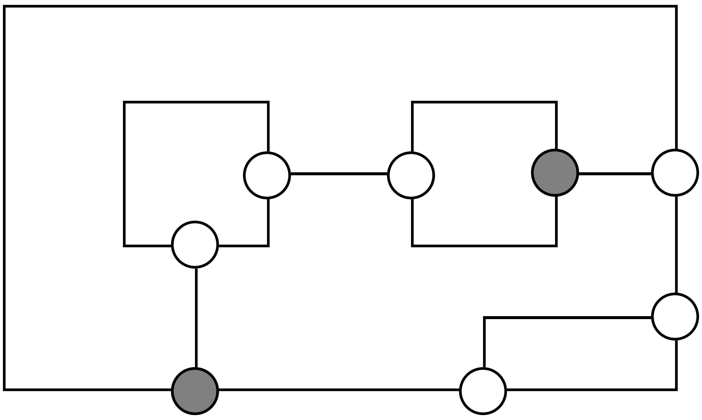
Directionality Constraints

The following diagram shows examples of bindings with legal directionalities.



Legal Bindings

Whereas the following diagram shows examples of illegal bindings:



Illegal Bindings

Connectivity Constraints

∂arwin allows bindings to fan-in. Bindings that fan-out (multi-casting) however, are illegal. The intention is to support fan-out in a future ∂arwin with a new "interface set" abstraction.

Typing Constraints

In addition to the directionality and connectivity constraints, ∂arwin also features an interface typing system. This system is independent of the typing systems of individual primitive component programming languages. Hooks are normally provided to implementors to access ∂arwin type constraint information.

Interface names in ∂arwin can be augmented with a type constraint which can be used to ensure that bound interface names are “type-compatible”. ∂arwin includes a type inference system to alleviate the need for tedious repetition of typing information, and to support ∂arwin generics.

Type constraints are placed in angle brackets after the interface name and separated either by whitespace or by comma. Type constraints can either be bound to a value or be left unbound.

Type constraint values are denoted by strings either explicitly by enclosing the value of the constraint within double-quote characters e.g. “fileT” or for alphanumeric strings, implicitly by omitting the enclosing double-quote characters e.g. fileT. Note that in this example fileT is equivalent ot "fileT".

Unbound type constraints are denoted by an identifer enclosed by an extra < > pair, e.g. <F>.

Example:

provide X < stream printf, “char *”, <data > >

Summary

∂arwin Reference

This section acts as an informal (and currently incomplete) reference to the ∂arwin programming language.

Syntax

The syntax of ∂arwin is given as in the extended form of traditional BNF used in the ISO Pascal Standand, except we use the metasymbol **xyz** (show bolded) as an alternative to the metasymbol "xyz" to represent terminal symbols. UPPERCASE letter are not significant in non-terminal meta-identifiers, but serve as commentary.

Special Symbols and Word Symbols

The ∂arwin configuration language vocabulary consists of special symbols, identifiers, numbers and strings.

special-symbol = .

word-symbol = **array** | **bind** | **component** | **const** | **dyn** | **export** | **forall** | **import** | **inst** | **provide** | **require** | **sizeof** | **when** .

It is planned to make the following “true” word symbols.

int | **char** | **double**

Comments

The constructs:

/* *any sequence of characters not containing a* */

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 not be nested. Nested comments can be obtained with the C pre-processor, e.g.

```
#ifdef COMMENT
....
#endif
```

¶ Comments follow those in C++

Identifiers

Identifiers are used as names of components, parameters, constants, interface names, 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 range identifier extends from its defining point (the point at which it is first introduced) to the end of the construct in which it is defined. The scope of other identifers extend from their defining point to the end of the component description.

Identifiers may include break characters "_".

id = (letter |"_") { letter | digit | "_" } .

Identifiers may be of any length. All characters 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 significant.

¶ Identifiers follow those in C++

Expressions

Expressions are a subset of those found in the C programming language, and are evaluated as in C.

expression = *a subset of expressions in the C programming language*

¶ Grammar of expression syntax -- Still To Do .

Standard Types

The following types are predefined in ∂arwin: **int**, **double** and **char** and **char ***. These types take their meaning from those found in C.

Composite Components

Composite components specify a component type possibly with formal parameters from which component instances can be created. A composite component configuration declares instances of component types, and describes the interconnection of these instances by the bindings between them.

Composite components are the building blocks of ∂arwin.

component = **component** COMP-id formal-parameters base-components
[dec-block]

dec-block = "{" declaration ";" "}"

declaration = array-declaration | const-declaration | interface-declaration |
guarded-declaration | forall-declaration | inst-declaration | bind-declaration

Identifiers declared within a component are not known within other composite components unless they are named within an interface declaration or for instances, unless the components acts as base component for a derived component.

The component type identifier is used to identify the component type within instance declarations, base component declarations and within generic component parameters.

Parameters

Parameters fall into one of two categories - normal parameters or generic parameters.

formal-parameters = ["(" parameter { "," parameter } ")"]

parameter = type PAR-id ["=" DEFAULT-expression] |
    component COMP-id ["=" DEFAULT-COMP-id]

type = "int" | "double" | "char" | "char *"

Normal Parameters

Normal parameters denote simple scalar values and are used within expressions, e.g. within interface declarations to set the upper bound of *provide*d or *require*d array identifiers, within guarded declarations to describe variant configurations or for evaluating the base case for recursive configurations, within *forall* declarations to iterate over a range of values, within *inst*ance declarations to index instances, within *bind* declarations to index instances and interface names, and as actual parameters to instances.

The value of a normal parameter is set by the corresponding actual parameter when the component type is instantiated.

Parameter Defaults

If a parameter default is supplied all subsequent component parameters must have a default also. Default-expressions are substituted into a component instance if no corresponding actual parameter is supplied.

¶ The scheme is similar to that in C++.

Generic Parameters

If a parameter begins with the special symbol **component** then the parameter is called a generic parameter. Generic parameters denote component types and can be used to declare instances of the type. The actual component type passed must conform to the formal parameter type, in both its parameters and interface identifiers.

¶ Must define component type compatibility rules.

Derived and Base components

A component can be derived from one or more (base) components. Derivation has the effect of including the configuration of the base component into the derived component, in particular the interface names and instances within the base component are visible within the derived component. If the base component has formal parameters, then actual parameters must be supplied unless a default value was specified.

base-components = [":" comp-def { "," comp-def }]

comp-def = COMP-ID arguments

arguments = ["(" expression { "," expression } ")"]

Constant Declarations

Constant declarations can be used to introduce names that denote specific constant expressions. Constants can be of the standard types.

const-declaration = **const** type id "=" expression { ";" type id "=" expression }

A constant identifier can only be defined once within a composite component.

Interface Declarations

Interface declarations serve to define a set of names that can be bound internally to encapsulated sub-interfaces or externally to the interface names of peers.

interface-declaration = (**provide** | **require** | **import** | **export**) interface-description

interface-description = interface-name type-constraints namespace

type-constraints = [“<“ constraint { [“,”] constraint } “>“]

constraint = (id | string) | “<<“ id “>>“

interface-name = id { “[“ INT-expression “]” }

namespace = [“@” STRING-expression]

Provide declarations list the names of identifiers(services) whose definition is being provided by this component to other components.

Require declarations list the names of identifiers(services) whose definition will be provided by other components to this component.

The *bind*ing declaration within the ∂arwin acts to "bind / assign /associate /link" the identifier provided by one component to an identifier required by another component.

Provided/required array names can be declared by suffixing the interface name with index expressions that specify the number of elements for each dimension of the array. The lower bound of an array identifer is always zero.

Export declarations list the names of identifiers(services) whose definition is being provided by this component to an external nameserver.

Import declarations list the names of identifiers(services) whose definition will be provided to this component by external components known to an external nameserver.

The namespace clause is only supported for import and export declarations. The interpretation of the namespace string is implementation dependent.

Exported/imported names are not visible at the enclosing darwin level.

Type constraints

Interface names can be augemented with one or more type constraints. Type constraints can be used to ensure that bound interfaces are “type-compatible”. Type constraints can either have a constant string value or be left unbound.

Constant type constraints are denoted by strings either explicitly by enclosing the constraint within double-quote characters e.g. “fileT” or for alphanumeric strings implicitly by omitting the enclosing double-quote characters e.g. <fileT>

Unbound type constraints are denoted by an identifer enclosed with an extra < > pair, e.g. <F>. Unbound type constraints become set whenever an associated interface identifier is bound to another interface identifer with a constant type constraint. If two unbound type constraints are bound then they become aliases for each other.

Instance Declarations

Instance declarations serve to create component instances from component types. Instances can be static or dynamic.

instance-declaration = **inst** instance { ";" instance }

instance = inst-def arguments location

inst-def= instance-name ":" [**dyn**] COMP-id

instance-name = INST-id { "[" expression "]" }

location = ["@" INT-expression]

Instantiation of a component type causes the interface names of the component type to be "created" for the component instance. These names are selected by prefixing the name of the component instance followed by a dot character.

If the component type has parameters, corresponding actual parameters must be specified, unless the component type has defaults. Actual parameters must correspond in order and type to the formal parameters of the component type.

Instance Arrays

Instance arrays can be created but must be defined before instantiation with an array declaration:

array-declaration = **array** inst-def { ";" inst-def }

Instance array elements are declared by suffixing the instance name with index expressions for each dimension of the array. Index expressions are of type integer.

Lazy Instances

Instances are normally created whenever the parent instance is created. Lazy instances however are only created when a bound provision is actually "called"/"invoked". This gives ∂arwin the potential for the lazy instantiation.

Dynamic Instances

See binding declarations below.

Distribution

The distribution of components can be specified with an integer location following the declaration. If the location is omitted the instance is co-located with its parent. By convention the top level instance is located at 0. The association of integer locations to actual processors/nodes is implementation dependent (see documentation on **red**)

Binding Declarations

Binding declarations are used to establish potential interactions between individual component instances either directly or indirectly via the interface of the component type. Binding can also be made to "dyn" components. Such bindings will cause a new instance of the bound component to instantiated each time the interface is “invoked”. Parameters to the new instance are suppiled from the invoking primitive component.

bind-declaration = **bind** binding { ";" binding }

binding = name “--” name

name = interface-name | instance-name "." interface-name | **dyn** component-name | **dyn** component-name “.” interface-name

Directionality Constraints

In ∂arwin, only the following forms of binding are allowed:

Binding	Binding Form
Normal (Peer)	**bind** *instance.requirement* -- *instance.provision*
Outbound	**bind** *instance.requirement* - - *requirement*
Inbound	**bind** *provision* -- *instance.provision*
Forwarded	**bind** *provision* -- *requirement*

Connectivity Constraints

∂arwin imposes the following constraints on the degrees of connectivity of bound requirements and inbound provisions:

forall x, x=*instance.requirement* | x=*provision*, 0 <=outdegree (x) <= 1

This states that fan-out (multi-casting) is not allowed. The intention is to support fan-out in a future ∂arwin with a new interface-set abstraction.

Type Checking

Whenever two interface names are bound, a type check also occurs on the corresponding type constraints, summarised below:

LHS Type Constraint	RHS Type Constraint	Type Checking Actions
Value	Value	compare LHS value & RHS value
Value	Unbound	define RHS constraint to LHS value
Unbound	Value	define LHS constraint to RHS value
Unbound	Unbound	alias LHS & RHS constraints

The first action ensures that the two interfaces are "type-compatible". Type compatibility is not determined by ∂arwin but rather by the underlying implementation platform. Different platforms will in general have different compatibity rules. If the underlying rules return an in-compatible result, the ∂arwin compiler will generate a compile-time error.

The remaining actions define rules for inferring one type from another:

- An unbound type constraint becomes set whenever the associated interface name is bound to another interface name with a type constraint value.
- If two interface names are bound which have unbound type constraints then the constraints become aliases for each other.
- Binding an aliased constraint to a value will cause its aliases to become bound to the value also. Similarly binding two aliased constraints will cause both sets of aliases to become one larger alias set.
- Essentially, the actions above a unification scheme for type constraints.

Guarded Declarations

Guarded declarations serve to conditionally elaborate one or more declarations. which are only elaborated if an integer expression evaluates to a non-zero value.

guarded-declaration = **when** INT-expression (declaration | dec-block)

Only the following declarations can be nested within a guarded declaration: instance declarations, bind declarations, forall declarations, and guarded declarations.

Example

```
when (row != n-1) || (col != 0)
        bind  processor [ row ][ col ].out -- processor [ (row+1)%n ][ col ].in;
```

Forall Declarations

Forall declarations serve to iteratively declare one or more declarations.

forall-declaration = **forall** id ":" INT-expression ".." INT-expression
(declaration | dec-block)

Only the following declarations can be nested within a forall declaration: instance declarations, bind declarations, forall declarations, and guarded declarations.

The scope of the forall identifier is restricted to the forall declaration block.

Example

```
forall row : 0..n-1  forall col : 0..m-1 {
   inst  processor [ row ][ col ] : t800;
   bind processor [ row ][ col ].out -- processor [ row ][ (col+1)%m ].in;
}
```

¶ ".." is horrible since it clashes with the declaration of C floating point numbers. This may cause problems with the C preprocessor.

Syntax Summary

word-symbol = **array** | **bind** | **component** | **const** | **dyn** | **export** | **forall** | **import** | **inst** | **provide** | **require** | **sizeof** | **when** .

id = (letter |"_") { letter | digit | "_" } .

expression = *a subset of expressions in the C programming language*

component = **component** COMP-id formal-parameters base-components [dec-block]

dec-block = "{" declaration ";" "}"

declaration = array-declaration | const-declaration | interface-declaration | guarded-declaration | forall-declaration | inst-declaration | bind-declaration

formal-parameters = ["(" parameter { "," parameter } ")"]

parameter = type PAR-id ["=" DEFAULT-expression] | **component** COMP-id ["=" DEFAULT-COMP-id]

type = "int" | "double" | "char" | "char *"

base-components = [":" comp-def { "," comp-def }]

comp-def = COMP-ID arguments

arguments = ["(" expression { "," expression } ")"]

const-declaration = **const** type id "=" expression { ";" type id "=" expression }

interface-declaration = (**provide** | **require** | **import** | **export**) interface-description

interface-description = interface-name type-constraints namespace

type-constraints = ["<" constraint { [","] constraint } ">"]

constraint = (id | string) | "<<" id ">>"

interface-name = id { "[" INT-expression "]" }

namespace = ["@" STRING-expression]

instance-declaration = **inst** instance { ";" instance }

instance = inst-def arguments location

inst-def= instance-name ":" [**dyn**] COMP-id instance-name = INST-id { “[“ expression “]” }

location = [“@” INT-expression]

array-declaration = **array** inst-def { “;” inst-def }

bind-declaration = **bind** binding { ";" binding }

binding = name “--” name

name = interface-name | instance-name "." interface-name | **dyn** component-name | **dyn** component-name “.” interface-name

guarded-declaration = **when** INT-expression (declaration | dec-block)

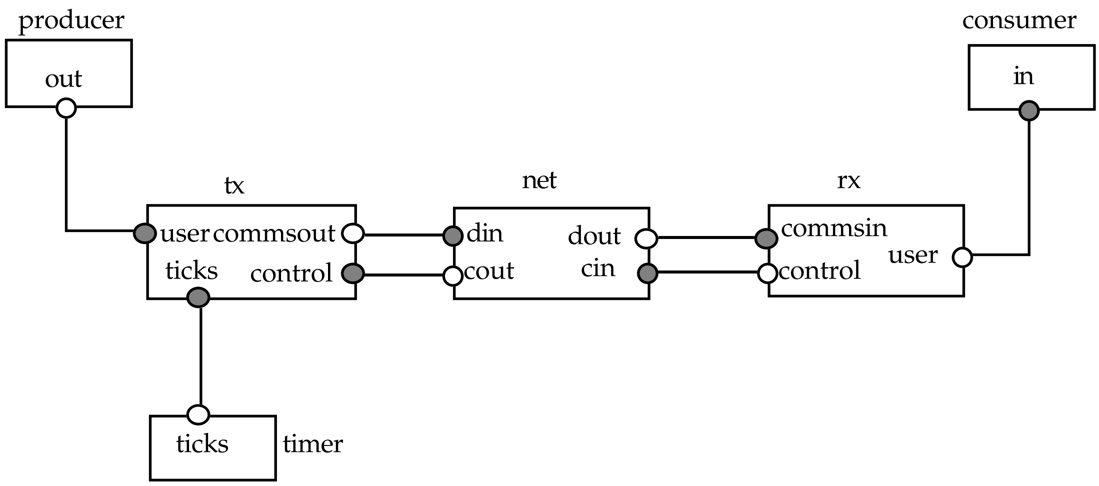
forall-declaration = **forall** id “:” INT-expression “..” INT-expression (declaration | dec-block)

∂arwin Examples

This section demonstrates the power and applicability of ∂arwin 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.

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:



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

#include “producer.dw”
#include “consumer.dw”
#include “sender.dw”
#include “receiver.dw”
#include “net.dw”
#include “timer.dw”

component Flowcontrol (**int** messages, **int** bias=5) {

inst consumer: Consumer;

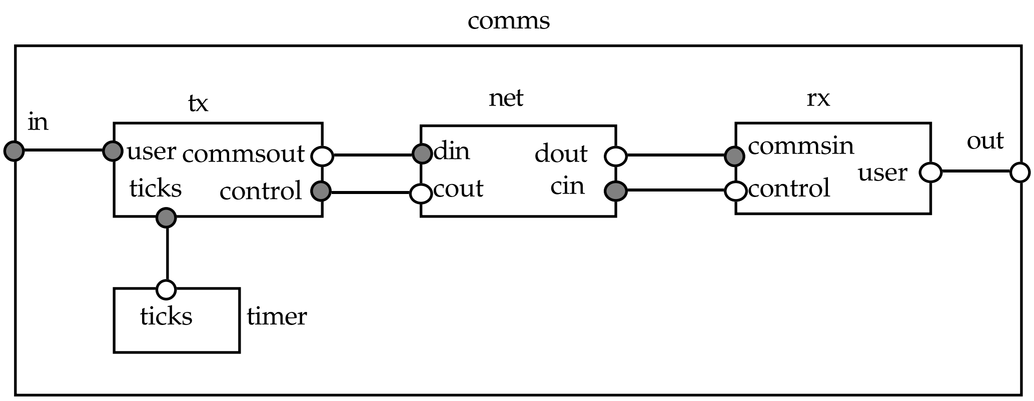
```
          producer: Producer (messages);
          tx:Sender ;
          rx:Receiver;
          net: Net (bias);
          timer : Timer;

bind      producer.out   – tx.user;
          timer.ticks    – tx.ticks;
          net.cout       – tx.control;
          tx.commsout    – net.din;
          net.dout       – rx.commsin;
          rx.control     – net.cin;
          rx.user        – consumer.in;
}
```

Apart from some parameterisation, the description is a straightforward encoding of the graphical form. The #**include**s select the component types required, the **inst** declaration instantiates a single instance of each, and the **bind** declaration 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 component, and parameterisation of components is allowed.

Note that the configuration description does not indicate whether a component is a primitive process component or a composite configuration component, nor does it indicate the types of bound interface names. This is also reflected in the graphical description which for space restrictions also omits component 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 component, the second to the net component. 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 composite component as in:

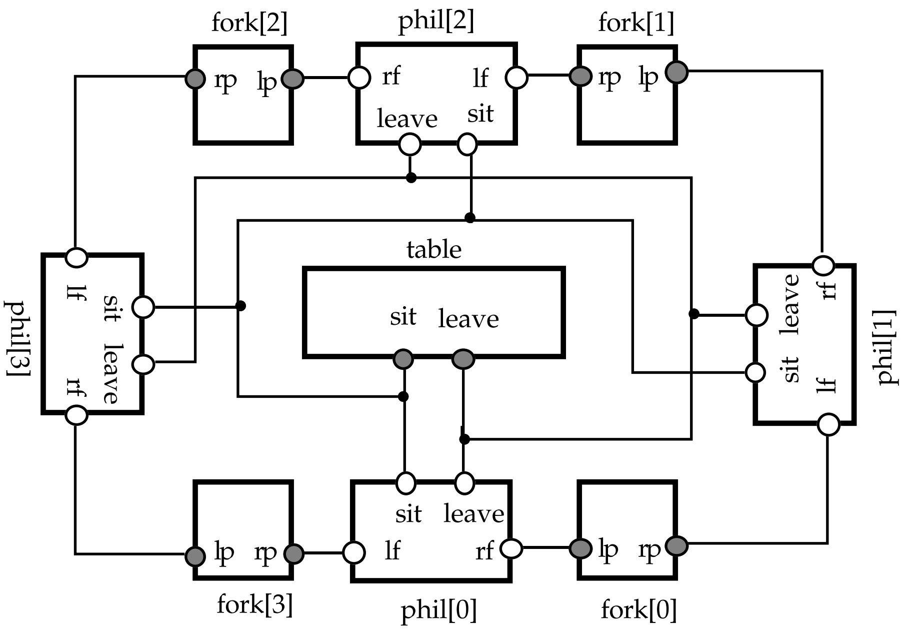


The use of hierarchic structuring is considered further in the third example.

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[1] 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 the following configuration structure:



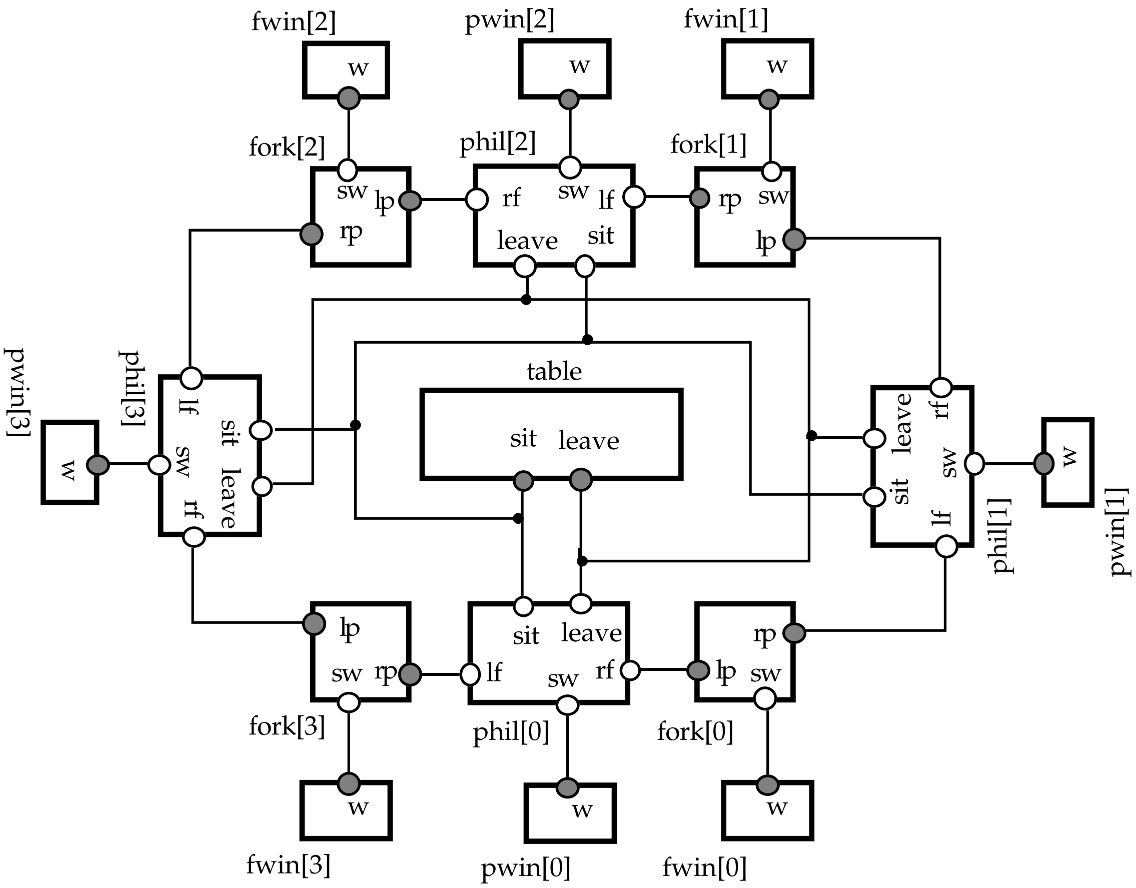
A configuration description for this structure is given below where lp=leftphil, rp=rightphil, lf=leftfork, rf=rightfork:

```
component Diners (int n=4) {
// assert n >=2

array phil[n] : Phil;
array fork[n] : Fork;
array pwin[n] : Window;
array fwin[n] : Window;

inst    table : Table(n);

forall k:0..n-1 {
        inst    phil[k] :phil (2000, 2000);
                fork[k]: fork;
        bind    phil[k].sittable        – table.sit;
                phil[k].leavetable      – table.leave;

                phil[k].rightfork       – fork[k].leftphil;
```

1 Prohibit starvation, free from deadlock and maximise parallelism.

```
                    phil[(k+1) % n].leftfork
                                          – fork[k].rightphil;

                // create and bind display windows for program

                inst    pwin[k] (0,(k+1)*14-13,(k+1)*14-5,8,17);
                        fwin[k] (0,(k+1)*14-5,(k+1)*14+1,8,17);
                bind    phil[k].std_write       – pwin[k].window;
                        fork[k].std_write       – fwin[k].window;
        }

        }
```

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** declarations act as universal quantifiers over the **inst** and **bind** declarations and range from 0 to the number of philosophers minus 1. The bound identifier k is used in expressions to bind the philosophers and forks in a ring, as well as in expressions for calculating the display coordinates for a set of window components. The window components 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 components is:

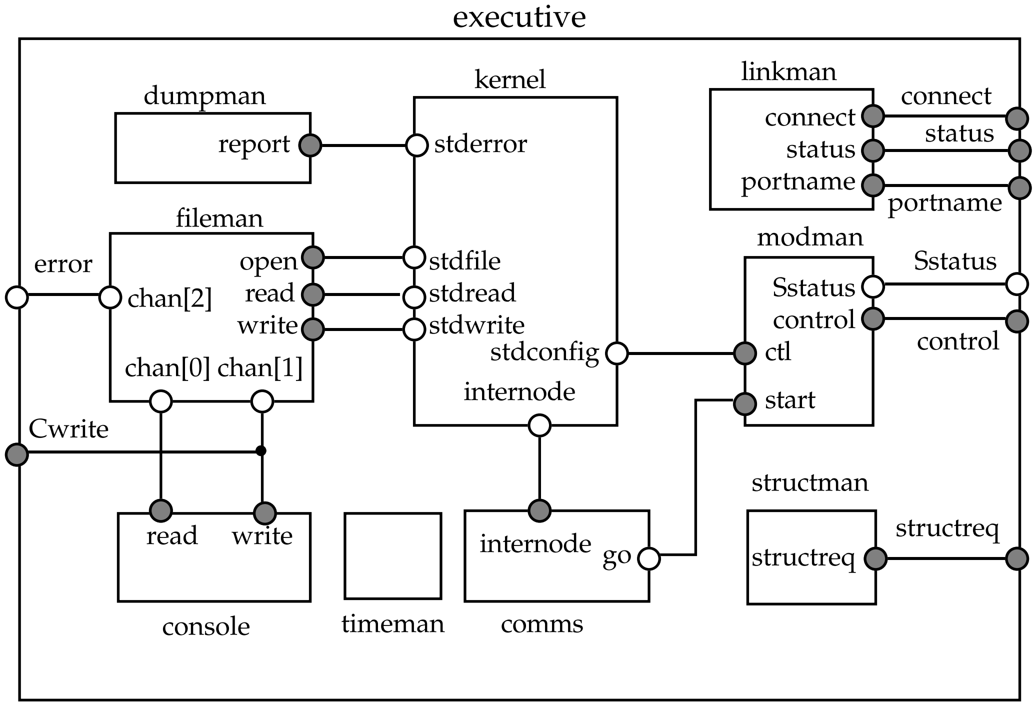


where lp=leftphil, rp=rightphil, lf=leftfork, rf=rightfork, w=window, and sw=stdwrite.

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

Run-Time Executive

The third example demonstrates the use of composite components for hierarchically structuring non-Regis programs. The example presents the overall structure of the 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



These components 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 communication (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: *stderror* for reporting errors in the process, ports *stdfile*, *stdread*, *stdwrite* for performing I/O and a *stdconfig* port for requesting dynamic configuration events. In order to reduce the profusion of binding that would be required to bind these ports for each process, the Conic kernel implements a default binding rule for these ports. Each standard port that is not explicitly bound is automatically bound to whichever ports the corresponding kernel standard port is linked to e.g. for the kernel above every unbound *stdread* port would be bound to fileman.read. Again for simplicity the standard ports of Conic 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)

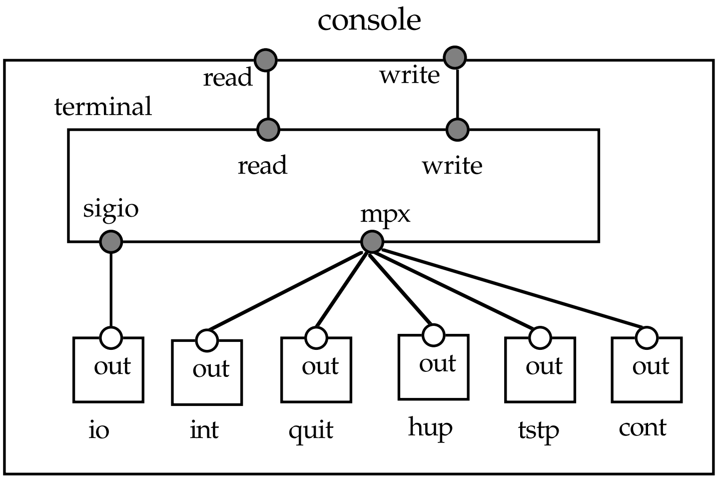
The configuration description for this executive is given below:

® Unix is a registered trademark of AT&T.

```
// #include's omitted
component Executive (int ticktime=1000);

inst    kernel : Kernel;
        timeman: Timeman(ticktime);
        dumpman : Dumpman;
        fileman: Fileman;

// configuration management components
inst    linkman : Linkman;
        modman : Modman;

require  Sstatus;
provide  connect; status; portname; control;

bind    connect             – linkman.connect;
        status              – linkman.status;
        portname            – linkman.portname;
        control             – modman.control;
        modman.Sstatus      – Sstatus;

// Internode Communication subsystem
inst    comms : Comms;

bind    kernel.internode    – comms.internode;
        comms.go            – modman.start;

// Console subsystem
inst     console : Console;

require  error ;
provide  Cwrite;

bind    fileman.chan[0]     – console.read;
        fileman.chan[1]     – console.write;
        fileman.chan[2]     – error;
        Cwrite              – console.write;

// Structure Query Handling
inst    structman : Structman;

provide  structreq;
bind    structreq           – structman.structreq;

// Define default links for tasks
bind    kernel.std_config   – modman.ctl;
        kernel.std_file     – fileman.open;
        kernel.std_write    – fileman.write;
        kernel.std_read     – fileman.read;
        kernel.std_error    – dumpman.report;
}
```

Two of the instances declared within this executive, console and comms, are in fact composite components 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:



The configuration description being:

#include "signalnums.h"
#include "terminal.dw"
component Console (**int** ttymode=0);

provide read; write;

inst terminal : Terminal (ttymode);

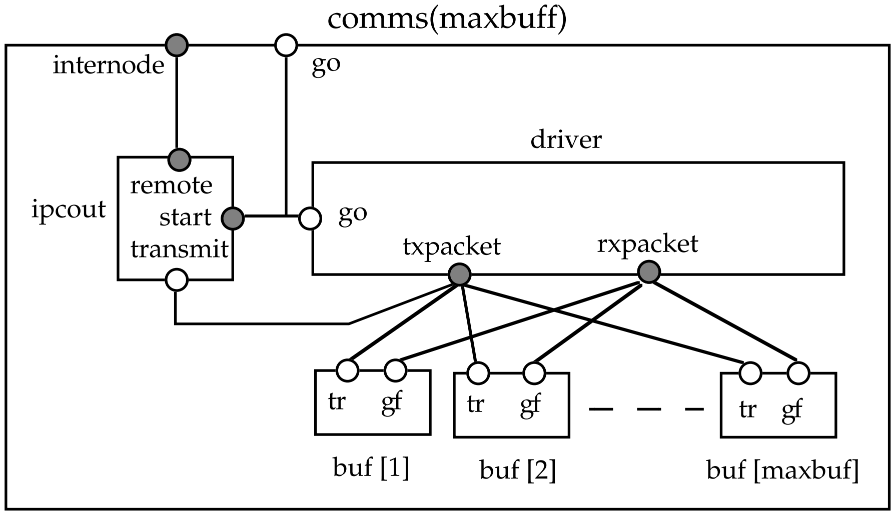
bind read – terminal.read;
write – terminal.write;

//Unix Signal handlers

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

bind io.out – terminal.sigio;
int.out – terminal.mpx;
quit.out – terminal.mpx;
hup.out – terminal.mpx;
tstp.out – terminal.mpx;
cont.out – terminal.mpx;
}

The comms composite component consists of a socket driver that utilises the socket facilities of BSD Unix [Leffler89] to perform non-local interactions. Messages destined for remote ports 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 reply 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:



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

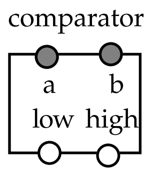
```
component Comms (int maxbuf=8);

inst    ipcout : Ipcout;
        driver: Socdriver;

array   buffer [maxbuf] : Ipcin

forall k:0..maxbuf-1 {
        inst    buffer[k]:Ipcin;

        bind    buffer[k].getframe    – driver.rxpacket;
                buffer[k].transmit    – driver.txpacket;
}

bind    ipcout.transmit         – driver.txpacket;

// external interface

provide  internode;
require  go;

bind    internode               – ipcout.remote;
        driver.go               – ipcout.start,go;
}
```

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

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 $O(\log_2 N)^2$.

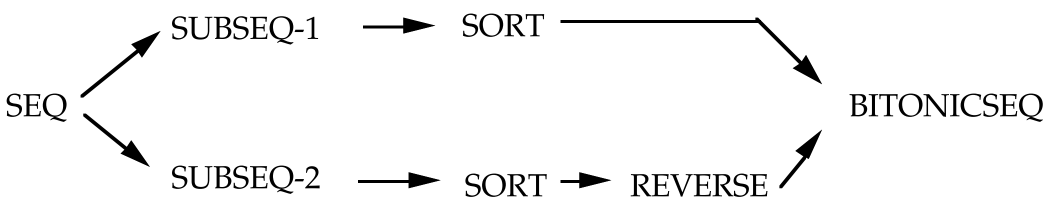
The algorithm uses a sophisticated network of primitive comparator elements (processes) that take 2 numbers as input and output the minimum and the maximum of the two numbers as outputs. In ∂arwin this can be represented by a component:



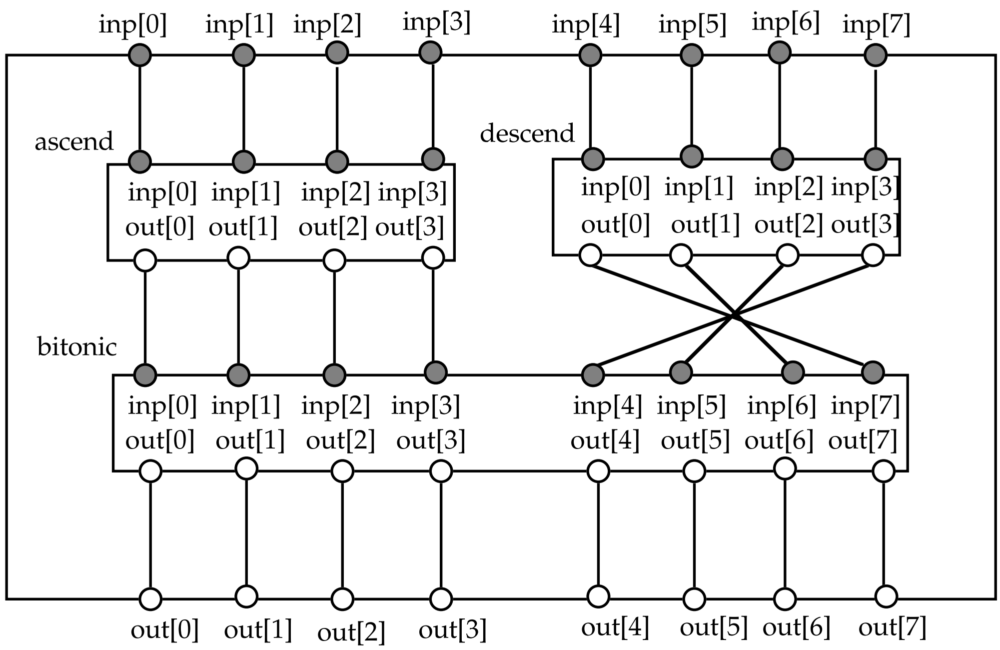
Batcher's sorting network takes an unsorted sequence of numbers, transforms this into a bitonic sequence, and then transforms the bitonic sequence into a sorted sequence.

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:



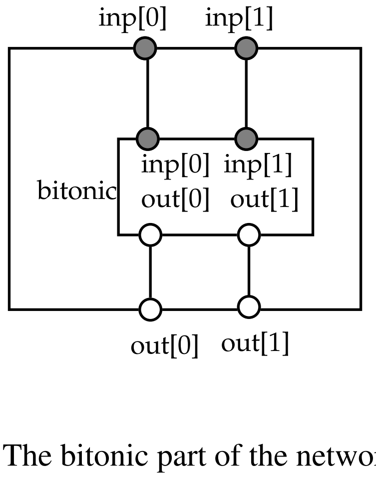
Thus for sorting 8 elements we would have the following sorting network:



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

```
#include  "bitonic.dw"
component Sort(int n);

provide  input[n] <<data>>;
require  output[n] <<data>>;

inst     bitonic : Bitonic(n);

when n>2 inst
         ascend:Sort (n / 2);
         descend:Sort (n / 2);

when n>2 forall k:0..(n / 2) -1 bind
         input[k]              – ascend.input[k];     // subsequence 1
         input[(n /2) +k]      – descend.input[k];    //subsequence 2

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

// base case
when n=2 forall k:0..1 bind
         input[k]              – bitonic.input[k];

-- always
forall k:0..n-1 bind
         bitonic.output[k]     – output[k];
}
```

This configuration illustrates (i) the declaration and use of arrays of composites ports, (ii) the use of multiple recursive instantiation, (iii) the use of guarded declarations to stop infinite recursion, (iv) the use of type constraints to ensure that inputs and outputs are of the compatible type. For the base case, n=2 we would have the following configuration structure for sort:



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

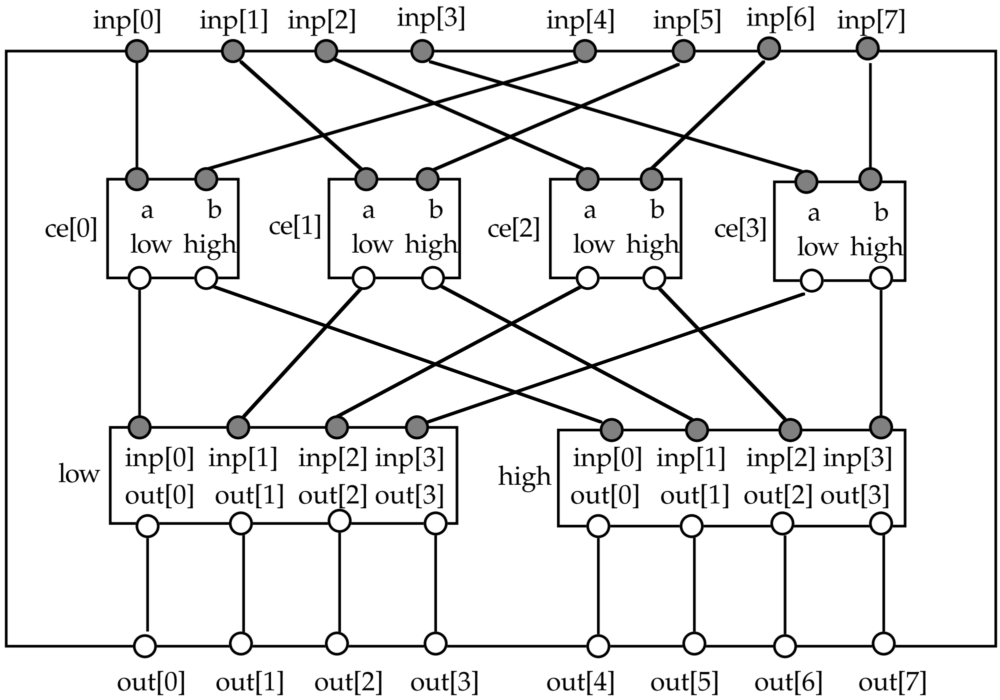
A network for sorting a bitonic sequence of 2n numbers a_1 to a_{2n}, can be constructed from n comparison elements and two bitonic sorters for n numbers. The comparison elements must form the two sequences

(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 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:

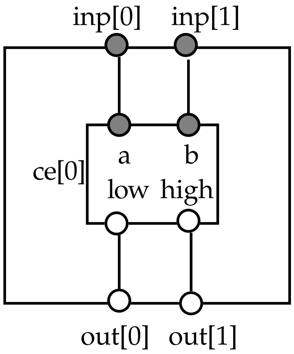


The configuration language description for bitonic networks is:

```
#include <comparator.dw>
component Bitonic(int n) {

provide  input[n] <<data>>;
require  output[n] <<data>>;

array  ce[n / 2] : Comparator;

forall k:0..(n / 2)-1 {
        inst    ce[k] : Comparator;
        bind    input[k]            -- ce[k].a;
                input[(n / 2)+k]    -- ce[k].b;
}

when n>2 inst
        low:Bitonic(n /2);
        high:Bitonic(n /2);

when n>2 forall k:0..(n /2)-1 bind
```

```
        ce[k].low              – low.input[k];
        ce[k].high             – high.input[k];
        low.output[k]          – output[k];
        high.output[k]         – output[(n / 2)+k];

when n=2 bind
        ce[0].low              – output[0];
        ce[0].high             – output[1];
}
```

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



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

```
component Batcher (int n=8);

inst    sort : Sort(n);
        data : Data(n);

forall k:0..n-1 bind
        data.out[k]            – sort.input[k];
        sort.output[k]         – data.inp[k];
}
```

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 ∂arwin. This example is also interesting in that the structure of the program is of far more interest and complexity than the algorithmic parts.

Parallel FFT

This next example is the design and implementation of a parallel algorithm to perform the Fast Fourier transform. The parallel algorithm is developed from the recursive sequential algorithm.

The Fourier Transform

Discrete Fourier Transform (DFT)

Following the conventions of [], the **N** -point discrete Fourier transform of a vector **Y** of complex numbers is the vector **F** defined by the formula:

$$F[p] = \frac{1}{N}\sum_{j=0}^{N-1} Y[j] * W^{jp} \quad F[p] = \frac{1}{N}\sum_{j=0}^{N-1} Y[j] * W^{jp} \qquad p = 0,1,\ldots,N-1$$

where the weighting factor **W** is defined as **exp(-2π/N),** the principal complex Nth root of unity. The inverse discrete Fourier transform is defined by:

$$Y[j] = \sum_{p=0}^{N-1} F[p] * W^{jp} \qquad j = 0,1,\ldots,N-1 \quad Y[j] = \sum_{p=0}^{N-1} F[p] * W^{jp} \qquad j = 0,1,\ldots,N-1$$

(2)

where the weighting factor **W** is defined as **exp(2π/N)**.

An algorithm which directly evaluated the discrete Fourier transform from the definition above would perform **O(n^2)** complex arithmetic operations. It should be noted that ignoring the **1/N** factor, the algorithm used to compute the DFT can be used to compute the inverse transform with **W** substituted for **W**.

Fast Fourier Transform (FFT)

The fast Fourier Transform as devised by Cooley and Tukey[2] is an **O(n log$_2$n)** method for exactly evaluating the DFT. The following development of the FFT algorithm is based on that outlined in [3].

Where **N** is even and ignoring the **1/N** factor, the DFT may be expressed by:

$$F[2r] = \sum_{J=0}^{M-1} SUM[j] * W_M^{jr} \qquad r = 0,1,\ldots,M-1$$

$$F[2r+1] = \sum_{J=0}^{M-1} DIFF[j] * W_M^{jr} \qquad r = 0,1,\ldots,M-1$$

(3)

(4)

where $M = N \div 2$

and **SUM[j] = Y[j] + Y[j+M]** (5)

and **DIFF[j] = (Y[j] - Y[j+M]) * W^j** (6)

Again ignoring the **1/N** factor, it can be seen that (3) and (4) have exactly the same form as (1). If **DFT_N(Y)** is the function which computes **F** the N-point Fourier transform of the vector **Y,** we can summarise (3) and (4) as shown below:

DFT_N(Y) = INTERLEAVE(DFT_M(SUM) , DFT_M(DIFF)) (7)

where **$M = N \div 2$**

and **SUM** and **DIFF** are computed by (5) and (6)

and **INTERLEAVE** forms **F** by placing the components of its first arguments in the even positions of **F** and the components of its second argument in the odd positions of **F**.

If **N** is an exact power of two, then (7) can be applied recursively with the base case of the recursion being:

DFT_1(Y) = Y[0] (8)

This is the basic idea behind the Fast Fourier transform, which except for [3] is usually well disguised. The next section describes the sequential program which implements this recursive algorithm.

Sequential FFT program.

Calculation of weighting factors.

At the first level of recursion the algorithm requires the first **M** powers of **W** (where **$M = N \div 2$**) i.e.

$$\exp(-2\pi/N)^{0} \;.\,.\,.\,.\,.\,.\,.\; \exp(-2\pi/N)^{M-1}$$

At the next level the first **M/2 -1** powers of **W_M** are required i.e:

$$\exp(-2\pi/M)^{0} \;.\,.\,.\,.\,.\,.\,.\; \exp(-2\pi/M)^{M/2-1}$$

However, **$\exp(-2\pi/(N/2)) = \exp(-2\pi/N)^2$**, consequently it is sufficient to compute the first **M** powers of **W** and at each successive stage of recursion choose the powers **2r** where **r = 0 . . . M/2-1.** Program 1, uses the array ***factor*** to store the powers of **W.**

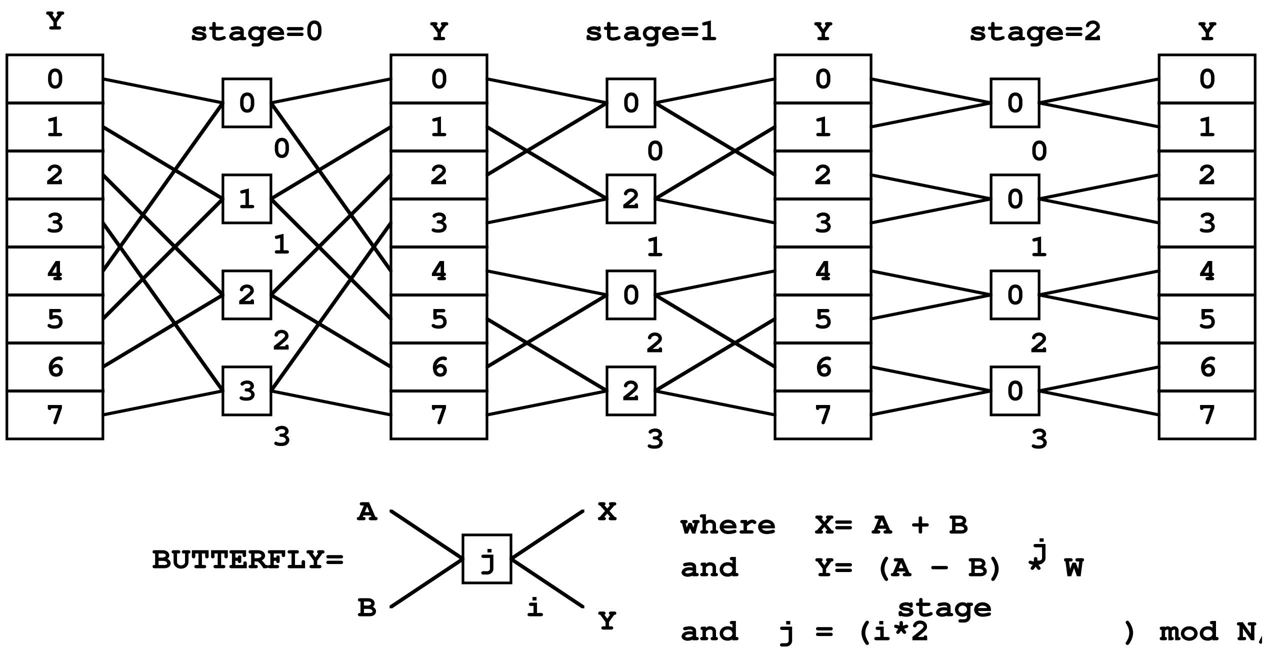
Interleaving

To allow the transform to be computed in situ in the input vector **Y**, the algorithm postpones interleaving to a single operation carried out at the end of the transform. The indices of vector elements to be exchanged are calculated by the function ***backwards***, which returns the bit-reversed value of its argument. A full explanation of the interleaving process may be found in [3].

The functions and types defined in ***compv*** and ***funcs*** may be found in Appendix 1., together with a test program which invokes the procedure ***FastFourier.***

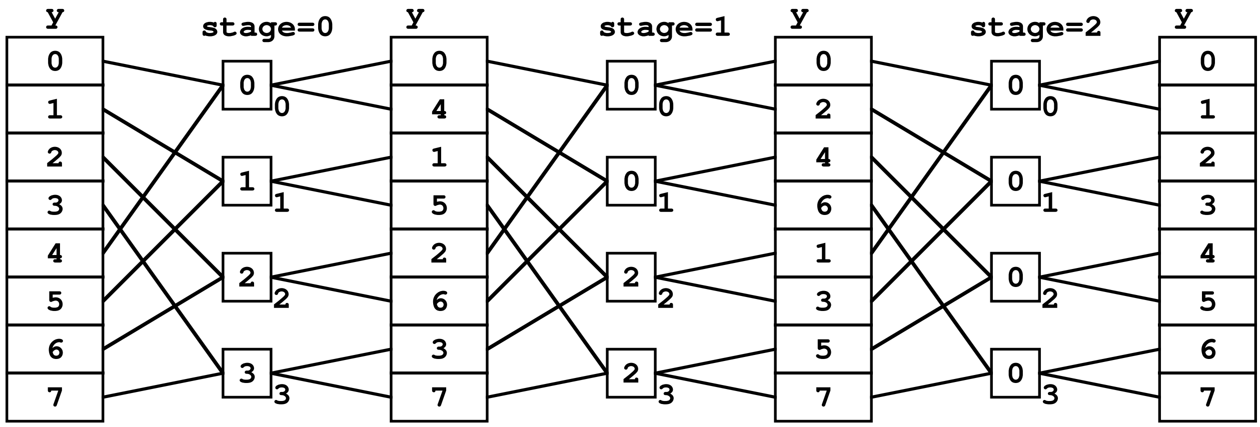
Parallel FFT program

Figure 1. is a graphical representation of data flow through the sequential FFT program for an 8-point input vector **Y**. It ignores the post- interleaving and scaling operations.

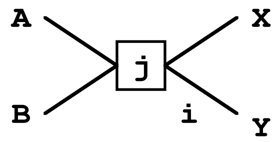


The data-flow diagram could be implemented directly as a parallel program with a process for each "butterfly". However, to execute this program in parallel, would require **$n\log_2(n)$** processors. Additionally, the interconnection between the butterfly elements is different for each stage. This would make the algorithm difficult to implement in VLSI.

If we regard each horizontal line of butterfly elements as being implemented on a single processor, the interconnection network between these processors can be shown to be a hybercube because each processing element is always pairing items , the indices of which, only differ in one bit position (see [4] for further details). However, as demonstrated in [5], a hypercube can be simulated by cycling through a perfect shuffle network $\log_2 N$ times, where N is the number of nodes in the hypercube. Figure 2 is Figure 1 redrawn with perfect shuffle interconnections.



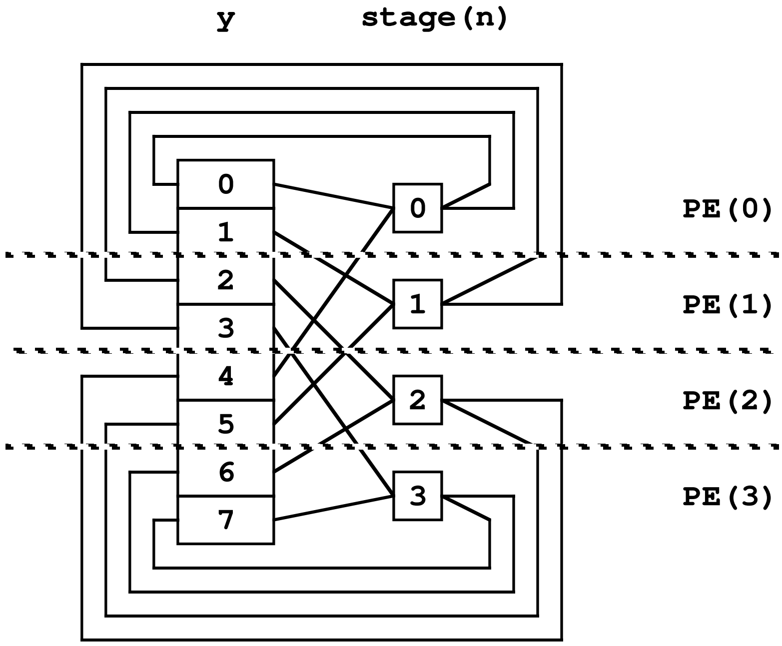
BUTTERFLY=



where $X = A + B$
and $Y = (A - B) * W^{j}$

and $j = (i \text{ div } 2^{stage}) * 2^{stage}$

Now if we take each horizontal line of butterfly elements and regard them as being executed by one processing element, the interconnection between processors will be a perfect shuffle network, rather than a hypercube switching network. The resulting parallel processor to perform the FFT will thus have the interconnection structure depicted in Figure 3. This architecture was originally presented in [5].



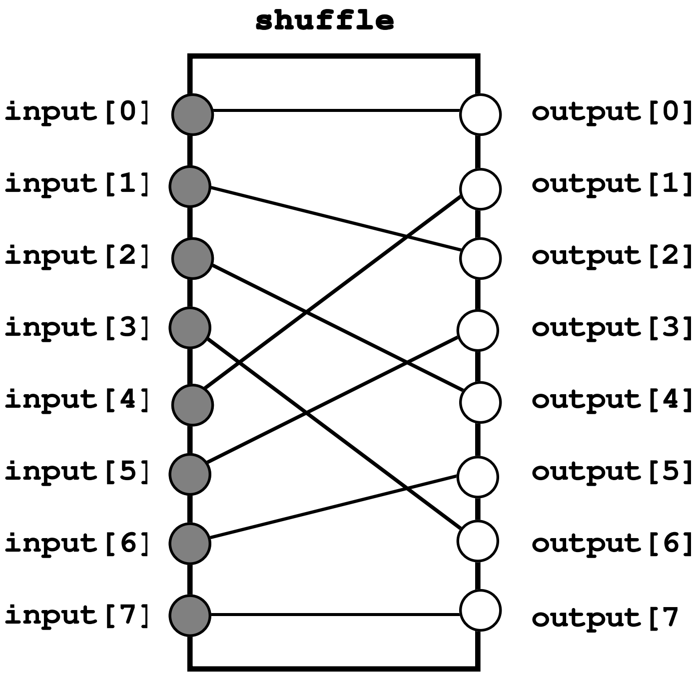
The data must be cycled through the processing elements (PEs) $\log_2 N$ times, at which point the DFT of the input will be available on the outputs in reverse binary order.

Implementation in ∂arwin

The implementation of Figure 3 as a Regis parallel program requires three components: a component to perform the butterfly operation, a perfect shuffle network and an interleaving network to order the output. To allow the program to operate in a pipeline (ie. to be cascaded with other programs) the implementation must adopt a suitable synchronisation strategy.

Perfect Shuffle Network

The perfect shuffle network connects the first half of its inputs to the even numbered outputs and the second half of its inputs to the odd numbered outputs. An 8-element perfect shuffle interconnection network:




```
component Shuffle(int n) {
provide  input[n] <<complex>>;
require  output[n] <<complex>>;

forall k:0..(n / 2)-1  bind
        input[k]               -- output[k*2];
        input[k+(n / 2)]       -- output[k*2 + 1];
}
```

Interleave Network

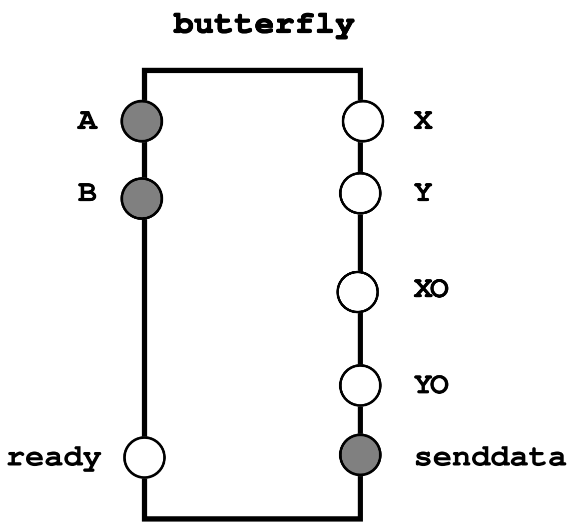
The interleave network performs the same function for the parallel algorithm as the interleave procedure provides for the sequential algorithm.. The network connects inputs to outputs in bit reversed order. The ∂arwin description of this network is:

```
component Interleave( int n) {


provide  input[n] <<complex>>;
```

```
require  output[n] <<complex>>;

forall k:0..n -1 bind
        input[k]        -- output[backwards(k,log2(n))];
}

#pragma verbatim
        extern int backwards(int, int);
        extern int log2(int);
#end
```

Butterfly

The butterfly task is the active component of the parallel algorithm. It performs the function described in Figure 2. A diagrammatic representation of the component which performs the function is shown in Figure 5. and the program is given in Program 4.



The task performs $\log_2 N$ operations and then presents the resulting values on its outputs **XO** and **YO** when it receives a signal on its entryport **senddata.**

Synchronisation

Each butterfly task sends a signal to its exitport **ready** when it is ready to perform another transform. The task shown in Program 5. combines these ready signals to indicate to the preceding stage that it can present a new set of input values. It this way stages can be pipelined. Effectively the **control** task performs an **and** function on the butterfly ready signals.

The complete parallel Fast Fourier algorithm can now be constructed from Programs 2-5, this is presented in Program 6.

```
component FFT (int n, int inverse) {
provide  input[n] <<complex>>;  senddata;
```

```
require  output[n]:<<complex>>; ready;

inst     shuffle : Shuffle (n);
         interleave : Interleave(n);
         control : Control(n);

forall k:0..(n / 2) -1 {
         inst     butterfly[k] (k,n,inverse);
         bind     // synchronisation links
                  butterfly[k].ready        -- control.senddata;
                  senddata                  -- butterfly[k].senddata;

                  // output links - interleave operation
                  butterfly[k].XO           -- interleave.input[k*2];
                  butterfly[k].YO           -- interleave.input[k*2+1];

                  // shuffle interconnection
                  shuffle.output[k*2]       -- butterfly[k].A;
                  shuffle.output[k*2+1]     -- butterfly[k].B;
                  butterfly[k].X            -- shuffle.input[k*2];
                  butterfly[k].Y            -- shuffle.input[k*2+1];
}

bind     control.ready              -- ready;

// hierarchic bindings
forall k:0..n -1 bind
         input[k]                   -- shuffle.input[k];
         interleave.output[k]       -- output[k];
}
```

FFT Conclusion

The use of ∂arwin allowed the decomposition of the algorithm into four constituent parts:

- shuffle network
- interleave network
- butterfly function
- synchronisation function.

Each of these components is implemented as a ∂arwin composite component or process component and finally connected together by a composite component to form the complete function.

Currently, the algorithm is a simulation in that it has been tested by executing it on a single processor where parallelism is simulated by interleaving. However, since ∂arwin supports mapping onto a variety of distributed computer architectures, it would be simple to execute the FFT program with real parallism. However, it is felt that the main reason for using ∂arwin is the development and testing of algorithms which may later be implemented in VLSI or on other special purpose hardware.

```
component Ft(int n) {
inst
         printer:Printer(n);
```

```
        fft:FFT(n, 0);
        sinewave:Sinewave(n);

forall k:0..n-1 bind
        sinewave.output[k]      -- fft.input[k];
        fft.output[k]           -- printer.print[k];

bind
        printer.ready           -- fft.senddata;
        fft.ready               -- sinewave.senddata;
}
```