# Think v4 (a.k.a. Nuptse)
# Programmer's Manual

October 19, 2009

# Revision History

| revision | date | description | author |
|---|---|---|---|
| 1.5 | 19/10/2009 | add documentation on multicast interfaces | Olivier Lobry (Orange Labs) |
| 1.4 | 15/07/2009 | add documentation on collection interfaces | Olivier Lobry (Orange Labs) |
| 1.3 | 15/07/2009 | add documentation on group interfaces<br>and garbage of dead components and services | Olivier Lobry (Orange Labs) |
| 1.2 | 06/07/2009 | add documentation on collection interfaces | Olivier Lobry (Orange Labs) |
| 1.1 | 24/12/2008 | add documentation on factories | Olivier Lobry (Orange Labs) |
| 1.0 | 01/09/2008 | initial version | Olivier Lobry (Orange Labs) |

# Contents

# Chapter 1

# Introduction

This document aims to give sufficient information to help developers *design* and *program* Think components and applications made of existing Think components. It does not intend to detail how to *use* the Think compiler in order to build a component-based software from existing component definitions. Such information can be found in the Think User's Manual.

Think is a native implementation of the Fractal component model[?]. It can be used to develop OS kernels though it is not restricted to this application domain: the framework can be used to develop any component-based system or application written in C. Thanks to the Fractal Component Model, Think adopts a clear separation between architecture and components. As a consequence Think accelerates native software development by allowing intensive re-use of predefined software component and rapid porting of infrastructure on new hardware targets. The Think project (http://think.objectweb.org) provides a compiler and several languages and mechanisms to design and program components.

Since its first design, Think has known several transformations through several versions. The latest version, Think v4, is known as Nuptse and focuses on simplicity and efficiency:

- it simplifies the burden of the developers of think-based software by providing simplified languages, especially for developers of functional code. It enables enhancement and simplification;
- it makes possible the generation of very efficient software by providing the possibility to better master the flexibility power and implementation (and so the associated cost) of generated software.

The goal of this new version is to make Think usable to design and program components for embedded applications and systems by taking into account the resource constraints specific to this application domain. In particular we are targeting embedded application like Wireless Network Sensors (WSN). This document focuses on the

Nuptse version of Think.

A component library named Kortex is also hosted in the repository of the project[1]. Kortex includes many components, some of them being devoted to execution infrastructure and OS development (memory manager, interrupt handler, semaphores, runtime schedulers...). Functional code can be written in C extended with reserved names representing architectural artifacts.

This document is split into two chapters. The first one details basic programming concepts and languages to design and implement basic components, that is, that should cover the requirements to design and implement, say, 90 % of components. The second chapter gives additional information on advanced concepts and languages to program control interfaces, develop and specify non-architectural aspect through properties and global extensions, understand optimizations, etc.

[1] For historical reason, this library is hosted by the Think project in the svn repository but may be extracted from it in a near future.

# Chapter 2

# Basic programming

## 2.1 Basic concepts

A *component*, in the Fractal sense, is a runtime entity, often called a *component instance*. A component has a type, called a *component type*, that defines the type of interactions the component can make. An *interface* is an interaction point, either an entry point, called a *server interface* or a *provided interface*, or an exit point, called a *client interface* or a *required interface*. In Fractal a component type is defined by the types of the interfaces that the component provides and requires. A component has a *content* and a *membrane*, the membrane having control over the content. The content may consist of *sub-components* or implementation code[1]. A component may also have *attributes* that reify non-functional properties of the component (like the speed of a serial port or the size of a buffer). The provided interfaces implemented by the content are called *functional interfaces* whereas the interfaces implemented by the membrane are called *control interfaces*. The idea here is that the content implements functional aspects of the component, whereas the membrane implements the non-functional aspects and provides control over this non-functional aspects. One particular non-functional aspect is the architectural one. The Fractal model defines a set of interface types that may be implemented to control the architecture of a system at runtime, namely:

- the `Component` interface that gives access to the interfaces provided by the component;
- the `BindingController` interface, that gives access to the interfaces required by the component and allows to change bindings;
- the `ContentController` that gives access to sub-components, if any, and allows to change the content of the component (i.e. add remove sub-components);
- the `AttributeController` that allows to get and set the value of an attribute.

[1]Or both in the case of Think, see section 3.4 for more details.

Remember that Fractal is a *runtime* component model. This precision is important since many other component models are design-time models. Fractal doesn't care about the design phase. When talking about Fractal implementations like Julia or Think we are talking about frameworks that are able to generate components that, once loaded, will be Fractal-compliant components. However, frameworks like these, do care about the design phase and provide languages to design components and program, typically an Interface Description Language (IDL) and an Architecture Description Language (ADL). They also provide one or more compilers or tools to generate data that will help to create components (i.e. component instances) at runtime. In this design phase, we are out of the scope of the Fractal model, though there are similarities.

One particular concept that does not exist at runtime is the concept of *component definition*. A component definition is a component type (it defines the set of client and server interfaces) with additional information about how this component is implemented. In other component models this is often referred to as a *component implementation* of a component (or component type). In Think the concept of implementation rather refers to the code that implements server interfaces.

Think provides an IDL to express the types of interfaces, and an ADL to express component definitions. In the sense of programming languages what is defined in an IDL or an ADL file (respectively interface types and component definitions) is a type. This is very similar to Java classes and interfaces that are defined in Java files: they are types. But beware: a component definition which is a type in the ADL sense, should not be confused with a component type in the Fractal sense, since it also expresses how a component type is implemented.

Also, we may sometimes say that a component definition contains sub-components, but actually only components (that are component instances) can contain sub-components. To be precise, a component definition may contains declarations of references to sub-components. At runtime, instances of this component definition will contain references to (sub-)components. This is very similar to a Java class that contains a declaration of a typed reference: at runtime, objects (i.e. instances) of this class will contain references to other objects.

## 2.2 The Interface Description Language (IDL)

Think provides a language, called an Interface Description Language (IDL), to declare interface types. In the Nuptse version, this language extends the C grammar with few keywords. In brief, an interface type is defined by a set of methods and a set of constants. Methods are declared as C functions and constants are declared as typed variable with an initial value.

An interface is declared as follows:

```
package <packageName:DotName> ;

(typedefs <filePath:DotName> ;)*

interface <interfaceTypeName:Name> {
```

```
    (<methodDeclaration> ; |
     <constantDeclaration> ;)*

}
```

packageName is the name of the package that contains this interface type. Like in Java, it must corresponds to the path of the directory that contains the file into which this interface type is defined. methodDeclaration must be a valid C function prototype declaration and constantDeclaration must be a valid initialized C variable declaration. The C types used in the declaration of methods and constants can be primitive C types (ex: int, unsigned int, short, struct, ...) but also types defined in external files : either a global file containing types which scope is the whole system which can be specified using the global-typedefs-file option passed to the compiler, or using the typedefs keyword. This keyword specifies that a file must be parsed before parsing the content of the interface definition and search for type definitions. These types can then be used in method or constant declaration and also in code that implements server interface or that uses client interfaces of this interface type. Note that if the -prefix-IDL-typedefs compiler option is set to true, uses of this types in the implementation code must be prefixed with the path of the interface type, with dots replaced with underscores.

**Example** In the following example, the interface type foo.api.Foo is defined as a method bar and a constant string CONST_STRING which value is "hi". Method bar takes a parameter a of type aType which is defined in file foo/api/aFile.h.

```
package foo.api ;

typedefs aFile ;

interface Foo {
        char * CONST_STRING = "hi";
    unsigned int bar(aType a);
}
```

The content of file foo/api/aFile.h is given bellow:

```
typedef struct {
  int x;
} aType;
```

## 2.3 The Architecture Description Language (ADL)

### 2.3.1 Introduction

### 2.3.2 Keywords

#### 2.3.2.1 component

**Usage** Declares a component definition.

```
[ abstract ] component <compDefName:DotName>
        [ extends <extCompDefName:DotName> ] {
        ...
}
```

**Description** Declares a component definition named `compDefName`. This definition may extend another definition named `extCompDefName`. Extending a component definition is like inlining the whole content of the extended definition into the extending one. Abstract component definitions are component definitions that are not sufficiently defined or that are not fully functional to exist at runtime. That is, they can only be used to declare an abstract sub-component in a component definition (see 2.3.2.7 for more details). Also, an component that contains an abstract sub-component must be declared abstract.

**Example** The following code declares an abstract component definition named `here.is.bar`, and another (concrete) component definition named `here.is.foo` as an extension of `here.is.bar`.

```
abstract component here.is.bar {
  provides itfTypeA as itfa
  contains subCompX = subCompDefX
  binds this.itfa to subCompX.itfa
}

component here.is.foo extends here.is.bar {
  provides  itfTypeB as itfb
  requires  itfTypeC as itfc
  contains subCompY = subCompDefY
  binds this.itfb to subCompY.itfb
  binds subCompY.itfc to this.itfc
  binds subCompY.itfa to subCompX.itfa
}
```

The definition above of `here.is.foo` is equivalent to:

```
component here.is.foo {
  provides itfTypeA as itfa
```

```
  provides itfTypeB as itfb
  requires  itfTypeC as itfc
  contains subCompX = sumCompX
  contains subCompY = subCompDefY
  binds this.itfa to subCompX.itfa
  binds this.itfb to subCompY.itfb
  binds subCompY.itfc to this.itfc
  binds subCompY.itfa to subCompX.itfa
}
```

#### 2.3.2.2 provides

**Usage** Declares a provided (a.k.a server) interface in a component definition.

```
provides <itfType:DotName> as <itfName:Name>
```

**Description** Declares a provided (a.k.a server) interface named `itfName` of interface type `itfType`.

**Example** The following code declares, in a component definition `here.is.bar`, a provided interface named `foo` of interface type `here.is.Foo`.

```
interface here.is.Foo {
  void foo1(int a, int b);
  int foo2(char x);
}

component here.is.bar {
  provides here.is.Foo as foo
}
```

#### 2.3.2.3 requires

**Usage** Declares a required (a.k.a client) interface in a component definition.

```
'requires' <itfType:DotName> 'as' <itfName:Name>
        [ '[' [ <size:int> ]  ']' ] [ ( 'mandatory' | 'optional' ) ]
```

**Description** Declares a required (a.k.a client) interface named `itfName` of interface type `itfType`. A client interface may be declared as optional or mandatory (default is mandatory). Any mandatory interface of a component instance must be bound to a server interface. The build chain will complain about unbound mandatory interfaces and will consequently fail. If the interface is specified as an array then the interface is a collection interface. Besides if `size` is specified, then the size of the collection interface will be equal to `size`. Otherwise, the size of the collection interface will be adjusted according to the maximum index specified in a bindings going from this interface. Note that for extensible collection interfaces (see 3.1), `size` means the *initial* size of the interface.

**Examples** The following code declares, in a component definition `here.is.bar`, a required interface named `foo` of interface type `here.is.Foo`.

```
interface here.is.Foo {
  void foo1(int a, int b);
  int foo2(char x);
}

component here.is.bar {
  requires here.is.Foo as foo
}
```

The following code declares, in a component definition `here.is.bar`, a required interface named `foo` of interface type `here.is.Foo` of size 4.

```
interface here.is.Foo {
  void foo1(int a, int b);
  int foo2(char x);
}

component here.is.bar {
  requires here.is.Foo as foo[4]
}
```

#### 2.3.2.4 binds

**Usage** Declares a binding from a client interface to a server interface.

```
binds <clientCompName>.<clientItfName>[<index>]
      to <serverCompName>.<serverItfName>
```

**Description** Declares a binding from client interface `clientItfName` of component `clientCompName` to server interface `serverItfName` of component `serverCompName`. Symbols `clientCompName` and `serverCompName` must either refer to a subcomponent or be equal to 'this'. In the latter case, 'this' refers to the defined component and interface names are used to refer to internal dual interfaces. However, to ease programming, `clientItfName` must be the name of a server interface (its dual being a client interface) and `serverItfName` must be the name of a client interface (its dual being a server interface). If `clientItfName` is a collection interface, then the index must be given.

**Examples** The following code declares, in a component definition `X`, a binding from client interface `foo` of component `a` to server interface `foo` of component `b`. It also declares a binding from provided interface `barIn` to server interface `bar` of component a, and a binding from client interface `bar` of component b to client interface `barOut` of current component.

```
component X {
  requires here.is.Bar as barIn
  provides here.is.Bar as barOut
  content a = aDef
  content b = bDef
  binds a.foo to b.foo
  binds this.barIn to a.bar
  binds b.bar to this.barOut
}
```

The following code declares, in a component definition `X`, a binding from element 0 of client interface `foo` of component `a` to server interface `foo` of component `b1` and a binding from element 1 of client interface `foo` of component `a` to server interface `foo` of component `b2`.

```
component X {
  content a = aDef
  content b1 = bDef
  content b2 = bDef
  binds a.foo[0] to b1.foo
  binds a.foo[1] to b2.foo
}
```

#### 2.3.2.5 attribute

**Usage** Declares an attribute.

```
attribute <attType:Type> <attName:Name>
        [ "=" <value:Expression> [ const ] ]
```

**Description** Declares an attribute named `attName` of type `attType` in a component definition. An initial value may be specified. This will be the value of the attribute once the system initialized. If `const` is specified the attribute will be constant, that is, will keep its initial value ant will not be modifiable at runtime. Usage in the functional code may be replaced by the specified value, so that trying to assign it in the functional code will possibly lead to a compile-time error.

**Example** The following code declares three attributes in a component definition `here.is.bar`. `foo1` is an int and has no initial value, `foo2` is of type short and will be instantiated with 3 as initial value, `foo3` is a constant char attribute which value is 10 and `foo4` is a constant string attribute which value is ”hello world”.

```
component here.is.bar {
  attribute int foo1
  attribute short foo2 = 3
  attribute char foo3 = 10 const
  attribute string foo4 = "hello world" const
}
```

#### 2.3.2.6 assigns

**Usage** Assigns a value to an attribute of a sub-component.

```
assigns <subCompName:Name>.<attName:Name>
        "=" <value:Expression>
```

**Description** Assigns value `value` to attribute `attName` of sub-component `subCompName`. `subCompName` must be the name of a sub-component declared in the component definition (see 2.3.2.7). If the attribute was already declared with a value, the latter is overwritten with the new value.

**Example** The following code declares a component definition `here.is.foo` that contains a sub-component `subComp` of type `here.is.bar` and assigns a new value to its attribute `att`.

```
component here.is.bar {
  attribute int att = 1
}

component here.is.foo {
        contains subComp = here.is.bar
        assigns subComp.att = 2
}
```

#### 2.3.2.7 contains

**Usage** Declares a sub-component in a component definition.

```
contains <subCompName:Name> ( : | = ) <compDef:DotName>
```

**Description** Declares a sub-component `subCompName` of component type `compDef` in a component definition. The ":" notation declares an abstract sub-component and must be used if and only if `compDef` is an abstract component definition. In that case, the enclosing component definition must also be declared abstract. Note however that an abstract component definition must not necessarily contains abstract sub-components.

**Example** The following example declares an abstract component definition `here.is.bar1` containing an abstract sub-component `c` which definition is the abstract component definition `here.is.foo1`, and a (concrete) component definition `here.is.bar1` that extends `here.is.bar1` and overloading the abstract sub-component `c` with a concrete sub-component which definition is `here.is.foo1`.

```
abstract component here.is.foo1 {
        ...
}

component here.is.foo2 extends here.is.foo1 {
        ...
}

abstract component here.is.bar1 {
    ...
        contains c : here.is.foo
}

component here.is.bar2 extends here.is.bar1 {
        contains c = here.is.foo2
}
```

#### 2.3.2.8 singleton

**Usage** Forces a component definition to be instantiated only once in a architecture.

```
singleton
```

**Description** Forces a component definition to be instantiated only once in a architecture. Two component instances of two different component definitions that contain a declaration of a sub-component which definition is declared as singleton will share the same sub-component instance at runtime.

**Example** The following code declares a singleton component definition `here.is.foo`, a component definition `here.is.bar1` that contains a sub-component `c1` of component type `here.is.foo` and a component definition `here.is.bar2` that contains a sub-component c2 of component type here.is.foo, and a sub-component c3 of component type here.is.bar1. In a instance x of the component definition here.is.bar2, x:c2 and x:c3:c1 refer to the same shared component instance.

```
component here.is.foo {
        ...
        singleton
}

component here.is.bar1 {
        contains c1 = here.is.foo
}

component here.is.bar2 {
        contains c2 = here.is.foo
        contains c3 = here.is.bar1
}
```

#### 2.3.2.9 content

**Usage** Specifies a file that contains implementation code.

```
content <fileName:DotName> [ raw ] )
```

**Description** Specifies that file which base name (i.e. without extension) is fileName with dot replaced with file separator, contains implementation code. The extension of the file name must be one of the following: ".c", ".s", ".S". If multiple files exist with the same base name, the first file that fits the mentioned extensions will be used, in the mentioned order. Note multiple files may be specified for implementing the component, so that implementation code can be split across several files.
If raw is specified, then the file contains code that does not directly implement server interfaces but usual C or assembly code instead and cannot make use of the model artifacts (access attributes, call client interfaces, ...).
Files will be searched in the component repository path list specified in the command.

**Example** In the following example, if we suppose that rep1 and rep2 are in the repository path list in that order, file rep1/a/b/f1.c contains code for the default implementation part, files rep1/a/b/c/f2.c and rep2/a/b/f3.c contain code for the implementation part imp1, file rep2/a/b/c/f4.c contains code for the implementation part imp2, and file rep2/a/b/d/f4.c is to be added as is[2]. Note that rep2/a/b/c/f2.c will be ignored.

```
component here.is.foo {
```

[2] See section 3.4 for explanations on multiple implementation parts.

```
        ...
        content a.b.f1
        content a.b.c.f2 for imp1
        content a.b.f3 for imp1
        content a.b.c.f4 for imp2
        content a.b.d.f5 raw
}
```

File structure:

```
rep1
 +- a
    +- b
       +- f1.c
       +- c
          +- f2.c
rep2
 +- a
    +- b
       +- f3.c
       +- c
          +- f2.c
          +- f4.c
       +- d
          +- f5.c
```

### 2.3.3 Deprecated Keywords and constructions

The following keywords and constructions are still supported by the compiler but are deprecated and may not be supported in future releases of the compiler. Programmers are strongly invited to use the corresponding keyword or construction.

| deprecated | new expression |
|---|---|
| composite | component[a] |
| primitive | component[2] |
| type | abstract component |
| attributes | attribute |
| attribute <attName> : <attType> | attribute <attType> <attName> |
| skeleton | content |
| ! nolcc[b] | provides fractal.api.LifeCycleController |
| implements | extends |

[a] There is no more distinction between primitive and composite components. Components are *hybrid* components in the sense that they can contain implementation code and sub-components

[b] ! nolcc means the absence of the keyword

## 2.4 The NuptC Component Programming Language

### 2.4.1 Introduction

Nuptse provides a Component Programming Language (CPL) called NuptC to write the functional code implementing the provided interfaces of a component definition. The CPL provides a way to the programmer to express the *mapping* between symbols defined in the ADL and IDL files corresponding to the component definition and the symbols in the C code that implements this component definition.

Contrary to previous CPLs of Think, NuptC has been designed in order to:

- minimize the burden of the programmers and clarify functional code by providing clear keywords representing the component concepts;
- allows optimizations by providing keywords that do not reflect particular implementation of the meta-data.

Functional code is parsed by the compiler and is translated into an Abstract Semantic Graph (ASG) using the CodeGen library[3]. Because NuptC does not extend the C grammar, files can be parsed with the C parser provided by CodeGen and the compiler implements a listener to handle the specific keywords. This ASG is then analyzed and transformed by the build chain and the resulting C files are then produced before being compiled by a C compiler (along with files containing the glue code).

Historically, NuptC was extending the C language with architectural-oriented reserved identifiers having well defined naming conventions. The grammar was the same but some identifiers were recognized as C mapping of architectural concepts. For example, to implement a method `foo` of a server interface `bar`, one had to declare a C function named `SRV_foo__bar` with appropriate parameters. While this approach simplified a lot the burden of programming component compared to the previous versions, and did allow arbitrary optimizations and implementations of architectural concepts, we went one step further: annotations.

CodeGen is able to parse annotations in C commentsdelimited by two "@@" tokens, like `/* @@ ServerMethod(foo, bar, bar) @@ */`. (Beware: never put annotations in document sections (i.e. starting with /**) or they will be silently ignored.) It does not define a particular syntax of the annotations. It just detects them and notifies them through a Java ParsingListener interface. The Nuptse compiler implements this Java interface to allow programmers to specify the mapping by annotating their C code. The main benefit of this new approach is that it is way less intrusive. First, programmers can decide to organize their mapping information as they want, either by annotating each C function, or by gathering all these information in the header or even in another file. But a greater benefit is encapsulation of legacy code. Indeed, it is very important, when transforming a legacy code to encapsulate it into a component, to modify the less possible the original code. Why there a many reasons for that, in is particularly important that subsequent evolutions of the original code can be applied to the transformed encapsulated code. Typically, if there a patch correcting a bug, it would be very interesting that the same patch can be applied to the encapsulated code.

[3]CodeGen is currently hosted by the Think project.

NuptC annotations allow to specify the names of the C symbols that represent architectural concept. Hence, it is possible, for example, to specify that the original name of a C function is used to represent a server method, so that there is no need to touch the original code to encapsulate it in a component definition. The *legacy* example in the example directory of the compiler gives a simple example of a C code that has been encapsulated in components without touching a single line of the original code.

The previous approach using naming conventions is however still supported. When programming components from scratch, some developers prefer this old approach because it makes mapping symbol more explicit, more ”visible”. To do this, NuptC defines implicit annotations that make available these old-style symbols[4]. These predefined annotations are detailed in the following sections.

In addition to annotations, NuptC provides some special keywords to get access to and check useful information related to architectural aspects. These keywords are described in section 2.4.3. Some other keywords are also available but rather to programmer controller. Those keywords are described in section 3.5.

Usually, instance artifacts can only be accessed or used from instance code. This means that for example a client method can only be called from a server or a private method. However, is some circumstances that make no ambiguity on the corresponding instance (for example if the code is not shared), they may be accessed/used from standard C code. See section 3.12 for more details.

### 2.4.2 NuptC annotations

#### 2.4.2.1 ServerMethod

**Usage** Declares a server method.

```
/* @@ ServerMethod(<serverInterfaceName >, <methodName>
          [, <functionName> ] ) @@  */
```

**Description** Specifies that the C function `functionName` implements the method `methodName` of the server interface `serverInterfaceName`. If the parameter `functionName` is omitted then the representing function is the following C function declaration (or definition). Note that the `methodName` parameter refers to the IDL definition of the type of the interface whereas the `serverInterfaceName` refers to the name of the interface provided by the implemented component, found in the corresponding ADL file.

**Examples** In the following example the programmer specifies that the C function `myBar` implements the method `bar` of the server interface `foo`.

```
// @@ ServerMethod(foo, bar, myBar) @@
[...]
int myBar(int x) {
```

[4]This behavior will be controllable in future releases through a compiler option

```
...
}
```

It the second following example the programmer specifies that the C function `bar` following the annotation implements the server method.

```
// @@ ServerMethod(foo, bar) @@
int bar(int x) {
...
}
```

**Predefined annotations** For each method `methodName` of each interface `itfName` provided by the component, Nuptse predefines the following annotation:

```
/* @@ ServerMethod(<itfName>, <methodName>,
      SRV_<itfName>__<methodName>) @@ */
```

#### 2.4.2.2 DefaultServerMethods

**Usage** Specifies that default symbol names will be used for representing server methods.

```
/* @@ DefaultServerMethods
          [ (<itfName1>, <itfName2>, ... <itfNameN>) ] @@ */
```

**Description** Specifies that the methods of server interfaces `itfName1` ... `itfNameN` are implemented by C function having the same names than the methods of the interface. If no interface is given, then this applies to all server interfaces of the component. Note that if two provided interfaces have methods with the same name, an error will be thrown if this annotation is used for both interfaces. In that case, the standard annotation `ServerMethod` must be used for at least one of the conflicting interfaces. This is a shortcut to writing the following annotation, for each method `methodName` of each interface `itfName` provided by the component:

```
/* @@ ServerMethod(<itfName>, <methodName>,
         <methodName>) @@ */
```

**Example** In the following example the programmer specifies that methods `bar` and `gnu` of interface `foo` are implemented by C functions with identical names.

```
// @@ DefaultServerMethods(foo) @@
[...]
int bar(int x) {
```

```
...
}
[...]
void gnu(short x) {
...
}
```

#### 2.4.2.3 ServerInterfacePrefix

**Usage** Specifies the prefix of functions that implement the methods of a server interface.

```
/* @@ ServerInterfacePrefix(<serverInterfaceName>,
        <prefix>) @@ */
```

**Description** Specifies that the methods of a provided interface `serverInterfaceName` will be implemented by C functions having the same name than the methods prefixed with `prefix`. This is a shortcut to writing the following annotation, for each method `methodName` of each interface `itfName` provided by the component:

```
/* @@ ServerMethod(<itfName>, <methodName>,
        <prefix><methodName>) @@ */
```

**Examples** In the following example the programmer specifies that the method `bar` and `gnu` of server interface `foo` are implemented respectively by C functions `my_bar` and `my_gnu`.

```
// @@ ServerInterfacePrefix(foo, my_) @@
[...]
int my_bar(int x) {
...
}
[...]
void my_gnu(short x) {
...
}
```

#### 2.4.2.4 ClientMethod

**Usage** Specifies the symbol representing a client method.

```
/* @@ ClientMethod(<clientInterfaceName>, <methodName>
        <functionName> ) @@
*/
```

**Description** Specifies that the C function symbol `functionName` represents the method `methodName` of the client interface `clientInterfaceName`. Calls to the client method can then be made by a C call using this symbol. Note that the `methodName` parameter refers to the IDL definition of the type of the interface whereas the `serverInterfaceName` refers to the name of the interface required by the implemented component, found in the corresponding ADL file.
Methods of a collection client interface are seen as arrays of the specified symbol. This means that calling method `methodName` of n$^th$ element of interface `clientInterfaceName` is done by doing `functionName[n]`.

**NOTE**: Please note that the expression given to index the collection interface **must not have side effects**. Typically, an collection interface element must not be done using something like `functionName[i++]`.

**Examples** In the following example the programmer specifies that the C function symbol `myBar` represents the method `bar` of the client interface `foo`. This method is called in function `aMethod`.

```
// @@ ClientMethod(foo, bar, myBar) @@

int aMethod(int x) {
   myBar(x);
}
```

In the following example the programmer specifies that the C function symbol `myBar` represents the method `bar` of the client collection interface `foo`. This method is called in function `aMethod` through element number 2 of the client interface.

```
// @@ ClientMethod(foo, bar, myBar) @@

int aMethod(int x) {
   myBar[2](x);
}
```

**Predefined annotations** For each method `methodName` of each interface `itfName` required by the component, Nuptse predefines the following annotation:

```
/* @@ ClientMethod(<itfName>, <methodName>,
      CLT_<itfName>__<methodName>) @@ */
```

#### 2.4.2.5 DefaultClientMethods

**Usage** Specifies that default symbol names will be used for representing client methods.

```
/* @@ DefaultClientMethods [ (<itfName1>, <itfName2>,
                                  ... , <itfNameN>) ] @@ */
```

**Description** Specifies that the methods of client interfaces `itfName1` ... `itfNameN` are represented by C function having the same names than the methods of the interface. If no interface is given, then this applies to all client interfaces of the component. Note that if two required interfaces have methods with the same name, an error will be thrown if this annotation is used for both interfaces. In that case, the standard annotation `ClientMethod` must be used for at least one of the conflicting interfaces. This is a shortcut to writing the following annotation, for each method `methodName` of each interface `itfName` required by the component:

```
/* @@ ClientMethod(<itfName>, <methodName>,
        <methodName>) @@ */
```

**Example** In the following example the programmer specifies that methods `bar` and `gnu` of interface `foo` are represented by C function symbols with identical names. These methods are called in function `aMethod`.

```
// @@ DefaultClientMethods(foo) @@

int aMethod(int x) {
  foo(x);
  bar(x+1);
}
```

#### 2.4.2.6 ClientInterfacePrefix

**Usage** Specifies the prefix of functions that represents the methods of a client interface.

```
/* @@ ClientInterfacePrefix(<clientInterfaceName>,
        <prefix>) @@ */
```

**Description** Specifies that the methods of a required interface `serverInterfaceName` are represented by C functions having the same name than the methods, prefixed with `prefix`. This is a shortcut to writing the following annotation, for each method `methodName` of each interface `itfName` required by the component:

```
/* @@ ClientMethod(<itfName>, <methodName>,
        <prefix><methodName>) @@ */
```

**Examples** In the following example the programmer specifies that the method `bar` and `gnu` of client interface `foo` are represented respectively by C functions `ext_bar` and `ext_gnu`.

```
// @@ ClientInterfacePrefix(foo, ext_) @@
[...]
int aMethod(int x) {
  ext_foo(x);
  ext_bar(x+1);
}
```

#### 2.4.2.7 Attribute

**Usage** Specifies the symbol representing an attribute.

```
// @@ Attribute(<attributeName>, <varName>) @@
```

**Description** Specifies that the C variable symbol `varName` represents the attribute `attributeName`. Calls to the client method can then be made by a C call using this symbol.

**Example** In the following example the programmer specifies that the C variable symbol `aVar` represents the attribute `att`. This attribute is accessed in function `aMethod`.

```
// @@ Attribute(att, aVar) @@
[...]
int aMethod(int x) {
   aVar = aVar + x;
   return aVar;
}
```

**Predefined annotations** For each attribute `attName`, Nuptse predefines the following annotation:

```
// @@ Attribute(<attName>, ATT_<attName>) @@
```

#### 2.4.2.8 DefaultAttributes

**Usage** Specifies that default symbol names will be used for representing attributes.

```
// @@ DefaultAttributes @@
```

**Description** Specifies that the attributes of the components are represented by C variable having the same names than the attributes. This is a shortcut to writing the following annotation, for each attribute `attributeName`:

```
// @@ Attribute(<attributeName>, <attributeName> @@
```

**Example** In the following example the programmer specifies that attributes `att1` and `att2` are represented by C variable symbols with identical names. These attributes are accessed in function `aMethod`.

```
// @@ DefaultAttributes @@

int aMethod(int x) {
  att1 = att1 + x;
  att2 -= x;
  return att1 + att2;
}
```

#### 2.4.2.9 AttributesPrefix

**Usage** Specifies the prefix of variables that represent the attributes of a component.

```
// @@ AttributesPrefix(<prefix>) @@
```

**Description** Specifies that the attributes of the component are represented by C variables having the same name than the attributes, prefixed with `prefix`. This is a shortcut to writing the following annotation, for each attribute `attributeName` of the component:

```
/* @@ Attribute(<attributeName>,
          <prefix><attributeName>) @@ */
```

**Examples** In the following example the programmer specifies that the attribute `att1` and `att2` are represented respectively by C variables `att_att1` and `att_att2`.

```
// @@ AttributesPrefix(att_) @@

int aMethod(int x) {
  att_att1 = att_att1 + x;
  att_att2 -= x;
  return att_att1 + att_att2;
}
```

#### 2.4.2.10 PrivateData

**Usage** Specifies a C global variable as a private data.

```
// @@ PrivateData [(<variableName>)] @@
```

**Description** Specifies that the C variable `variableName` will be a private variable. Private variables are component variables, that is, each component instance of a component definition will have its own instantiation of the declared private variables. If parameter `variableName` is omitted then the annotation indicates that the variable declaration that follows is a private data.

**Examples** In the following example the programmer specifies that variable `aVar` is a private data which is accessed (as a normal C variable) in server method `bar`.

```
// @@ PrivateData(aVar) @@
[...]
int aVar;
[...]
// @@ ServerMethod(foo, bar) @@
int bar(int x) {
  aVar = aVar * x;
}
```

The following example illustrates the use of the annotation where the name of the variable is omitted.

```
// @@ PrivateData @@
int aVar;
[...]
// @@ ServerMethod(foo, bar) @@
int bar(int x) {
  aVar = aVar * x;
}
```

**Predefined annotation** Variable name `PRIVATE` is considered as a private variable. That is, Nuptse predefines the following annotation:

```
// @@ PrivateData(PRIVATE) @@
```

#### 2.4.2.11 PrivateMethod

**Usage** Specifies a C function as a private method.

```
// @@ PrivateMethod [(<functionName>)] @@
```

**Description** Specifies that the C function `functionName` is a private method. A private method is a function that can access data of the component instance (access attributes, call client methods, ...), like a server method, but does not implement any method of a server interface. If parameter `functionName` is omitted then the annotation indicates that the function declaration that follows is a private method.

**Examples** In the following example the programmer specifies that function `prvMeth` is a private method which makes a call to a client method `gnat`.

```
// @@ ClientMethod(gnut, gnat, gnat) @@
// @@ PrivateMethod(prvMeth) @@
[...]
int prvMethod(int x) {
  gnat(x * x);
}
[...]
// @@ ServerMethod(foo, bar) @@
int bar(int x) {
  prvMethod(x + 2);
}
```

The following example illustrates the use of the annotation where the name of the function is omitted.

```
// @@ ClientMethod(gnut, gnat, gnat) @@
[...]
// @@ PrivateMethod @@
int prvMethod(int x) {
  gnat(x * x);
}
[...]
// @@ ServerMethod(foo, bar) @@
int bar(int x) {
  prvMethod(x + 2);
}
```

**Predefined annotation** Functions that begin with `PRV_` are considered as a private method. That is, for such a function named `PRV_foo`, Nuptse predefines the following annotation:

```
// @@ PrivateMethod(PRV_foo) @@
```

#### 2.4.2.12 KeepName

**Usage** Indicates to keep unchanged a C function.

```
// @@ KeepName @@
```

**Description** Specifies that the name of the following C function should not be changed by the Nuptse compiler. This annotation can typically be used in case the function is to be called from assembly code that cannot be transformed by the compiler.

**Examples** In the following example the programmer specifies that the name of the C function `bar` representing the server method `bar`, should not be changed by the compiler.

```
// @@ KeepName @@
// @@ ServerMethod(foo, bar) @@
int bar(int x) {
  [...]
}
```

#### 2.4.2.13 IgnoreDeclarations

**Usage** Indicates to eliminate declaration of global variables from the C source code.

```
// @@ IgnoreDeclarations( <varName1>, ... , <varNameN> ) @@
```

**Description** Specifies that the variables named `varName1`, ..., `varNameN` should be eliminated by the compiler when transforming the source code. This is typically used when turning a C global variable into an attribute when encapsulating legacy code. Note: `IgnoreDeclarations` annotations may be implicitly deduced in the future from the `Attributes` or `AttributePrefix` annotations.

**Examples** In the following example the programmer specifies that the attribute `att` is represented by symbol `aVar` and that the declaration of the C global variable is to be ignored, because the variable has be turned into the attribute `att`.

```
// @@ Attribute(att, aVar) @@
// @@ IgnoreDeclarations(aVar) @@
  [...]
int aVar;

int bar(int x) {
  aVar = x;
}
```

### 2.4.3 Special keywords

#### 2.4.3.1 CLTID_

**Usage** Gets the value of the identifier of a client interface.

```
CLTID_<itfName>
```

**Description** The keyword will be transformed into an expression that represents the value of the identifier of client interface `itfName`.

#### 2.4.3.2 SRVID_

**Usage** Gets the value of the identifier of a server interface.

```
SRVID_<itfName>
```

**Description** The keyword will be transformed into an expression that represents the value of the identifier of server interface `itfName`.

#### 2.4.3.3 ATTID_

**Usage** Gets the value of the identifier of an attribute.

```
ATTID_<attName>
```

**Description** The keyword will be transformed into an expression that represents the value of the identifier of attribute `attName`.

#### 2.4.3.4 IS_BOUND

**Usage** Check whether a client interface is bound.

```
IS_BOUND( cltId )
```

**Description** The keyword will be transformed into an expression that is true (in the C sense) if the client interface identified by expression `cltId` (which can be calculated using a `CLTID_` expression) is bound to a server interface.

#### 2.4.3.5 CALL_PRV_METH

**Usage** Generic call of a private method.

```
CALL_PRV_METH( funcPtr )
```

**Description** The keyword can be used to call a a private method using the pointer `funcPtr` to this private method. This can typically be used to call a private through an array of pointers to private methods.

# Chapter 3

# Advanced Programming

## 3.1 Collection interfaces

As pointed in section 2.3.2.3, client collection interfaces can be declared as an array of an given interface type. By adding the `[extensible=true]`, the collection interface becomes an extensible collection interface in the sense that its size can be extended at runtime (that is, new interface elements can be added). Note that the size of an extensible collection interface is always greater or equal that its original size, that is, the compiler generates meta-data for all interface elements that correspond to the original size, and this meta-data will never be freed. Consequently, the user should take care of the initial size given to the extensible collection interface. In most cases, not specifying the initial size (and letting the compiler adjust it according to the encountered bindings) is the best solution.

A new binding controller is added that allows to handle extensible collection interfaces. It allows to access and add elements of collection interfaces using a "-¡suffix¿" suffix. For example, if `foo` is a collection interface then `"foo-12"` will refer to an element of interface `foo`. Elements are sorted alphabetically. When using numbers as suffices, users should take care to the fact that `"foo-12"` will be sorted before `"foo-2"` but after `"foo-02"`.

Note that indices used when calling a collection interface (see 2.4.2.4) have nothing to do with interface element suffices except that they respect the same order. This means that for example, if interface `foo` as two elements called `foo-0` and `foo-2`, calling method bar like `bar[0]`, will not necessarily result in calling interface element `foo-0`. However, if calling `bar[i]` results in calling `foo-0` and calling `bar[j]` results in calling `foo-2`, then necessarily $i < j$.

## 3.2 Multicast interfaces

Multicast interfaces are collections interfaces, but which elements can be called all (or several) at once. Roughly, one has to specify its client interface as a collection interface (extensible or not). If this interface is annotated with the multicast=true property, the component is transformed at load time so that the interface appears as an unicast interface from inside but still multicast from outside. The unicast call is bound to a code that does the transformation into a multicast call. This code is generated in the backend by a specific builder (objectweb.think.primitive.nuptse.controllers.multicast.MultiCastInfo).

From an architectural point of view, if the programmer provides:

```
component foo {
  requires Bar as bar[] [multicast=true, extensible=true]
  content fooImplem
}
```

then he/she gets:

```
component foo {

  // implem of multicast (generated in the backend)
  implementation MultiCast [shared=false,
     builder=objectweb. ... .multicast.MultiCastInfo]

  // "hidden" entry point of multicast implem
  internal provides Bar as barInMc
        in MultiCast [single=true]

  // "exit" point of the multicast implem, same
  // name and same properties than the original one
  // so that it replaces the original one from outside
  // point of view
  requires Bar as bar[]
        in MultiCast [extensible=true]

  // original interface has been transformed
  // into a "hidden" unicast interface still has
  // the same name so that annotations in
  // functional code are still valid
  internal requires Bar as bar

  // functional code is untouched
  content fooImplem

  // inline binding to the multicast implem
  // so that the result is as efficient as if it has
  // been coded by hand
  selfbinds default::bar to MultiCast::barInMc
        [static=true, inline=true]

  // "hidden" multicast control interface
  // implemented in MultiCast implem
```

```
  internal provides fractal.api.MulticastController
    as barInMc-ctrl in MultiCast [single=true]

  // a "hidden" and "inlined" client interface
  // interface is added to allows the functional
  // code control the behavior of the multicast
  // interface
  internal requires fractal.api.MulticastController
    as bar-ctrl

  selfbinds default::bar-ctrl to MultiCast::barInMc-ctrl
       [static=true, inline=true]

}
```

"Hidden" internal interfaces are interfaces that do not appear in the component type but only at implementation level. They are not be visible from outside (compile time and runtime (i.e through BindingController)). To bind them, they must be prefixed with the implem name. Two implems may have interfaces with same names, provided at least one of them is "hidden".

A multicast control interface (of type fractal.api.MulticastController) is also added for each multicast interface. Its purpose is to allows the functional code control the cast behavior (either multicast or broadcast) and the multicast set.

The backend will generate the following private data:

```
// @@ PrivateData @@
// cast mode (true=multicast, false=broadcast)
_Bool multicastMode = false;

// @@ PrivateData @@
// pointer to an array that implements the multicast set
_Bool multicastSet;

// @@ PrivateData @@
// size of the array
multicastSetSize;
```

for each void method of the multicast interface:

```
void meth(<params>) {
  int index;
  if (multicastMode) {
    for (index = 0; index < multicastSetSize; index++) {
        if (multicastSet[index])
          client_meth[index](<args>);
    }
  } else {
```

```
      for (index = 0; index < <taille_itf >; index++) {
          client_meth[index](<args >);
      }
    }
}
```

and for each returning method:

```
<type> meth(<params>) {
    int index;
    if (multicastMode) {
      for (index = 0; index < multicastSetSize; index++) {
          if (multicastSet[index])
            client_meth[index](<args >);
      }
    } else {
      for (index = 0; index < <taille_itf >; index++) {
          client_meth[index](<args >);
      }
    }
    return <defaultValue(<type>)>
}
```

values between '<>' being of course generated according to the architectural properties.

## 3.3 Factories

Factories can be used to dynamically generate an instance of a given component definition. Fatories provide the `fractal.api.Fractal` interface that provides a `instanciate` method. They also require the memory.api.Allocator interface in order to dynamically allocate memory. The component definition of factories is parametrized with a component definition from which a factory is desired. A "factorized" component must provides the `fractal.api.ComponentIdentity` interface.

To create a new factory component of a given component definition, just declare in you adl file as :

```
component aComponentDefinition {

        // a component that requires the
        // fractal.api.Factory interface
        contains aFactoryUser = ...

        // an allocator, like unix.memory.lib.malloc
        // nedeed by the factory
        contains anAllocator = anAllocatorDefinition
```

```
        // the factory itself, parametrized with
        // the desired component definition
        contains myFactory = fractal.lib.factory(fooDef)

        // binds factory to allocator
        binds myFactory.allocator to anAllocator.allocator

        // binds factory user to factory
        binds aFactoryUser.factory to myFactory.factory
}
```

Then to dynamically create a new instance and call one of its interface:

```
// @@ ClientInterfacePrefix(factory, FACT_) @@
// @@ ClientInterfacePrefix(dynItf, DYN_ITF_) @@
// @@ ClientInterfacePrefix(compId, CI_) @@
// @@ ClientInterfacePrefix(selfBC, SELFBC_) @@

void aMethod() {
        any itf;
        // create a new component instance
        // the instanciate method returns
        // the component-identity server
        // interface of the new instance
        any newCompCI = FACT_instanciate();

        // the returned interface must be bound
        // to a required fractal.api.ComponentIdentity
        // interface in order to get the desired interface
        BC_bind(CLTID_compId, newCompCI);

        // get the desired server interface and bind
        // it to the corresponding client interface
        newCompSrvItf = CI_getInterface("aServerItf");
        BC_bind(CLTID_dynItf, newCompSrvItf);

        // now the the client interface can be called,
        // from here and from anywhere in the code
        DYN_ITF_aServerItf();
}
```

## 3.4 Implementation Parts

In the Fractal model, a component may contain sub-component or code (that compose its content), and a primitive component is defined as follows: *"A component that does not expose its content, but has at least one control interface, is called a primitive component"*. To our understanding, this definition does not prevent from having components made of sub-components *and* implementation code. The approach taken in Think is to actually make no distinction between primitive and composite components, since the property of being a primitive component is derived by other properties: the fact that the component provides or not a `ContentController` interface. A server interface can be either implemented by a sub-component or directly by implementation code. Another particularity of Think, is that the implementation code of a component may be made of multiple implementation parts. That is, the set of services provided by a component may be split between multiple parts, each responsible of a subset of the provided interfaces. One important outcome of this choice is that different properties may be associated to different implementation parts. For example, it is possible to specify that the code implementing a control interface is shared between components, while the code that implements a functional interface (or another control interface) is not, or by specifying different builders for different implementation parts (see section 3.11).

When programming a component the basic way, programmers do not have to care about implementation parts. For this reason, the Nuptse compilers defines a hidden implementation part named `default` with the following rules:

- any provided interface that is not bound to a sub-component or is not implemented by another implementation is implemented by the default implementation;
- any client interface can be called by the default implementation part;
- any content file which is not specified for a particular implementation part contains code for the default implementation.

The ADL provides the following keywords to specify required interfaces, provided interfaces and content files for a particular implementation part. Note that the current notation is subject to change in the future. In the following example, the programmer specifies that the provided interface `foo` is implemented by the implementation part `anImplementationPart` that is made of the content file `aFile` and may call the required interface `bar. here.is.Foo`.

```
component here.is.bar {
  [...]
  provides foo as here.is.Foo in anImplementationPart
  requires bar as here.is.Bar in anImplementationPart
  content aFile for anImplementation
}
```

## 3.5 Programming controllers

Think provides a way to specify and implement control interfaces. The approach taken in the Nuptse version is to make almost no distinction, in the specification, implementation and compilation, of control and functional interfaces. The only difference is that programmers of implementations of control interfaces can use ”privileged” keyword to get access to meta-data generated by the compiler. (This is very similar to the fact that, in operating systems, kernel code has accessed to privileged machine instruction that user code do not.) In addition, the compiler distribution comes with a set of implementations of the standard Fractal interfaces. This approach has the following benefits:

- it provides an uniform way to the programmers to specify and implement interfaces, whatever they are control or functional interfaces;
- it makes possible to specify, implement and use new control interfaces, or program new implementation of the Fractal interfaces;
- all optimizations that can be applied to functional interfaces can be applied the same way to control interfaces. Also, it is possible to control the implementation of meta-data corresponding to control interfaces as it is possible for functional interfaces.

As mentioned above, NuptC provides ”privileges” symbols for programming implementation of control interfaces. This keywords give a way to initialize implementation code with values concerning the architecture known at compile time and access and modify meta-data at runtime. All these keywords start with the `META_` prefix.

### 3.5.1 Client Interfaces Meta-Data

`META_NB_CLT_ITFS` Compile-time value representing the number of client interfaces of the component.

`META_CLTITF_TABLE` Variable name that declares a table containing the name and the id of each required interface. This table must be declared as an array of any type, which size must be at least the number of client interfaces. The type of the table is transformed into an array of struct with two fields:

- `itfName` the name if the interface
- `itfId` the id of the interface

The table is initialized with values known at compile time. For example if a component requires two interfaces `foo` and `bar`, then the following declared variable:

```
any META_CLTITF_TABLE[META_NB_CLT_ITFS];
```

is transformed into the following code:

```
struct {
  char * itfName;
  any itfId;
} META_CLTITF_TABLE[META_NB_CLT_ITFS] = {
  { "foo", <fooId> },
  { "bar", <barId> }
}
```

where `fooId` and `barId` are the identifiers of respectively the `foo` and `bar` client interfaces.

`META_CLT_ITF_SET` Runtime function to set the server interface identifier corresponding to a given client interface identifier.

```
void META_CLT_ITF_SET(any cltItfId, any srvItfId);
```

`META_CLT_ITF_GET` Runtime function to get the server interface identifier corresponding to a given client interface identifier.

```
any META_CLT_ITF_GET(any cltItfId);
```

### 3.5.2 Server Interfaces Meta-Data

`META_NB_SRV_ITFS` Compile-time value representing the number of server interfaces of the component.

`META_SRVITF_TABLE` Variable name that declares a table containing the name and the id of each provided interface. This table must be declared as an array of any type, which size must be at least the number of server interfaces. The type of the table is transformed into an array of struct with two fields:

- `itfName` the name if the interface
- `itfId` the id of the interface

The table is initialized with values known at compile time. For example if a component provides two interfaces `foo` and `bar`, then the following declared variable:

```
  any META_SRVITF_TABLE[META_NB_SRV_ITFS];
```

is transformed into the following code:

```
struct {
  char * itfName;
  any itfId;
} META_SRVITF_TABLE[META_NB_SRV_ITFS] = {
```

```
  { "foo", <fooId> },
  { "bar", <barId> }
}
```

where `fooId` and `barId` are the identifiers of respectively the `foo` and `bar` server interfaces.

### 3.5.3 Attributes Meta-Data

`META_NB_ATTS` Compile-time value representing the number of atributes of the component.

`META_ATT_SET` Runtime function to set the value of an attribute given its identifier.

```
void META_ATT_SET(any attId, any attValue);
```

`META_ATT_GET` Runtime function to get the value of an attribute given its identifier.

```
any META_ATT_GET(any attId);
```

### 3.5.4 Components Meta-Data

`META_NB_SUB_COMPS` Compile-time value representing the number of sub-components of the component.

`META_SUBCOMP_TABLE` Variable name that declares a table containing the name and the id of each sub component. This table must be declared as an array of any type, which size must be at least the number of sub components. The type of the table is transformed into an array of struct with two fields:

- `compName` the name of the sub component
- `compId` the id of the sub component

The table is initialized with values known at compile time. For example if a component contains two sub components `foo` and `bar`, then the following declared variable:

```
  any META_SUBCOMP_TABLE[META_NB_SUB_COMPS];
```

is transformed into the following code:

```
struct {
  char * compName;
  any compId;
} META_SUBCOMP_TABLE[META_NB_SUB_COMPS] = {
  { "foo", <fooId> },
  { "bar", <barId> }
}
```

where `fooId` and `barId` are the identifiers of respectively the `foo` and `bar` sub components.

## 3.6 Properties

The Think ADL gives the possibility to attach properties to declarations of the different artifacts of the architecture model: components, client interfaces, server interfaces, attributes, implementation and content files. The syntax is:

```
[<propName>=<propValue >]
```

where `propName` and `propValue` are strings. Properties give a way to pass information to the different parts of the compiler.

For example, the property `binds foo.i to bar.i [static=true]` specifies that the binding between `foo.i` and `bar.i` will not change at runtime. The builder of the client interfaces of `foo` (that is responsible of the generation of code that makes the mapping between the identifier of a client interface and the identifier of the server interface it is bound to), can take this property into account and generate direct calls for invocations of the methods of `foo.i`.

```
// attached to a component
component ... [<propName>=<propValue >] {
        // attached to a server interface
        requires ... [<propName>=<propValue >]
        // attached to a client interface
        provides ... [<propName>=<propValue >]
        // attached to an attribute
        attribute ... [<propName>=<propValue >]
        // attached to a binding
        binds ... [<propName>=<propValue >]
        // attached to a content file
        content ... [<propName>=<propValue >]
        // attached to an implementation part
        implementation ... [<propName>=<propValue >]
}
```

### 3.6.1 Existing properties

Some properties are already defined and taken into account by the default builders of the Think compiler. These properties are listed bellow, with the default value.

**`implementation ... [shared=true|false]`** Specifies that the code of the implementation part is shared between component instances.

**binds ... [static=true|false]** Specifies that the binding will not change at runtime. Default value is `false`.

**binds ... [inline=true|false]** Specifies that, in case the binding is static, the compiler should try to inline the code that implement the server interface into the code of the caller of the client interface, or generate inline directives so as to inform the C compiler to inline the code.

**attribute ... [const=true|false]** Specifies that the attribute will not change at runtime. Default value is `false`.

**provides ... [single=true|false]** Specifies that, in the scope of the super component, this entry point will be the only one for the corresponding interface type. Default value is `false`.

**provides ... [no-dynamic-binding=true|false]** Specifies whether no client interface will be dynamically bound to this server interface. Default value is `false`.

**[attribute=<value>]** Can be attached to `component`, `attribute`, `provides`, `requires` or `implementation` to add `__attribute__(<value>)` to glue descriptors of respectively components, attributes, server interfaces, client interfaces and implementation. This can typically be used to control the placement of glue descriptors in memory sections (rom, ram, flash, ...).

**[embedded-desc=true|false]** TO BE COMPLETED...

## 3.7 Aspect Oriented Programming using Global Extensions Mechanism

The Architecture Description Language introduced in section 2.3 allows to define a component by extending another definition. Starting from a initial definition it is possible to add new interfaces, new attributes or new subcomponents. It is also possible to specify new properties to existing interfaces, attributes, components, ... In the following, the component definition `staticComp` extends the definition `comp` by making static the declared interfaces.

```
component staticComp extends comp {
  binds x.i to y.i  [static=true]
  binds x.j to z.j  [static=true]
}
```

Nuptse introduces the notion of *global extension* as a way to extend multiple component definitions in a AOP-like manner using pattern matching. An extension specification is a standard ADL file but where names can have ”jokers” that will be matched against names found in an architecture description. For example applying the following extension definition make all bindings static of any component definition.

```
component **.* {
  binds *.* to *.* [static=true]
}
```

The list of global extensions is specified with the `ext-files`. Users can also specify the option `ext-path` to specify directories where to find extension files. The compiler will first look in the `ext-path` then in the `src-path` when looking for global extensions. Each global extension is matched against each component definition found in the architecture given as input to the build chain, and it matches the original definition is extended accordingly.

Note that currently pattern expression is very limited: only ”**.*” or exact names are supported at the moment. More possibility (based on Java pattern matching) will be available in the future. Also note that this not possible to specify a path to a component instance (only names of component definition is supported at the moment). This will be implemented in a near future. These two limitations do not however invalidate the principles of the described approach.

## 3.8 Group Interfaces

A group interface can be used to gather a set of interfaces. This is a convenient feature for components (like an OS component) that provide a set of services and when you do not necessarily want to express precise dependencies. Group interfaces are a language facility that it transformed into normal interfaces through architecture transformations (there is no impact on the build phase). In the following example, component ’OS’ provides interface ’os’ of type ’api.OS’, and component ’app’ requires interface ’os’ of the same type. ’app.os’ is bound to ’OS.os’

api.OS is defined as follows:

```
package api;
group OS {
        api.LibC as libc
        api.Thread as thread
}
```

Interface ’os’ of ’app’ (resp. ’OS’) is transformed into os-thread of type ’api.Thread’ and os-libc of type ’api.LibC’. Also binding from ’app.os’ to ’OS.os’ is transformed into bindings from ’app.os-libc’ to ’OS.os-libc’, and from ’app.os-thread’ to ’OS.os-thread’. Hence:

```
component app {
```

```
    ...
    requires OS as os
    ...
}

component OS {
     ...
     provides OS as os
     ...
}

component system {
     ...
     contains app = app
     contains OS = OS
     binds app.os to OS.os
     ...
}
```

is equivalent to:

```
component app {
    ...
    requires api.LibC as os-libc
    requires api.Thread as os-thread
    ...
}

component OS {
     ...
    provides api.LibC as os-libc
    provides api.Thread as os-thread
     ...
}

component system {
     ...
     contains app = app
     contains OS = OS
     binds app.os-thread to OS.os-thread
     binds app.os-libc to OS.os-libc
     ...
}
```

Consequently, functional dependencies can also be directly expressed more precisely. For example, if we know that component 'app' does not need a 'thread' interface, we can directly write:

```
component app {
    ...
    requires api.LibC as os-libc
    ...
}
```

With garbage enabled (see section 3.10.1), server interface OS.os-thread will be removed, and implementation code will be eliminated using gcc -02.

This feature then provide an uniform way to provide a set of services without having to bother of what will be actually used. An OS component can provides a 'libc', 'stdio', 'thread', ... services through a 'OS' group interface. Applications can expressed rough 'OS' or fine dependencies 'OS-libc', 'OS-thread', ... and only services that are actually used will remain in the resulting OS.

## 3.9 ADL predicates

The ADL defines some predicates to condition some declarations.This predicate are particularly useful in case of global extensions because they can be applied to different component definition having different structure. For example, it should be useful to add a `BindingController` interface to a component only if it has client interfaces. This is expressed using the `hasCltItf` attribute as in the following example:

```
component **.* {
  provides fractal.api.BindingController
    as binding-controller in BindingController
    if hasCltItf
}
```

## 3.10 Compilation control and optimizations

### 3.10.1 Architectural optimizations

If option "-garbage" of the compiler is set to "true", then unused services are garbaged at load time. This means that:

- functional server interfaces that are not bound are removed;
- components that have no server interfaces anymore (or just a fractal.api.LifeCycleController interfaces) are removed;
- if the server interface is provided by a composite component, the dual internal interface and the outgoing bindings are removed;

- if the server interface is directly implemented by NuptC code, the corresponding server methods are converted into private methods, that can then be garbaged by the C compiler with appropriate options (like -O2 of gcc).

Garbage of a server interface can be disabled by setting its 'garbage' property to 'false':

```
provides api.Foo as foo [garbage=false]
```

Garbage of all interfaces can be disabled by applying the global extenions 'no-garbage' (like using the option '-ext-files=no-garbage'.

### 3.10.2 Architectural-oriented meta-data generation

Programmers can control the generation of meta data using properties and use global extensions to apply the same properties to a set of components, interfaces, ... and separate this specification from the description of the architecture. Using the default builders, what can be currently controlled is:

- the placement of data using the `attribute` property (see section 3.6.1);
- the organization of descriptors using the `embedded-desc` property (see section 3.6.1);

Additionally the following optimizations are done:

- direct call are generated for static bindings and no meta-data is generated for the client interface in the `META_CLTITF_TABLE` table.
- for code that is proper to a single component instance, the addresses of meta-data are known at compile-time, so the "this" parameter that may be generated for implementations of server methods is not used;
- for single implementation of server interfaces, the "this" parameter is not generated for implementations of server methods and the "this" argument is not calculated when calling such a method. Combined with the previous optimization, the "this" parameter disappear completely;
- the organization of descriptors using the `embedded-desc` property (see section 3.6.1);

Note that not using the "this" parameter in not-shared implementations is distinguished to the not generation of the "this" parameter of single interfaces. This is necessary because a client interface bound to a server interface which implementation is not shared may possibly be re-bound to another server interface which implementation is not single and that expects a valid "this" parameter to function properly. Only if the server interface is single, the "this" parameter need not to be passed because this means that there is (and will be) no other server that implements the same interface type and so client interfaces that are bound to it cannot be bound to another server interface (provided type correctness of bound interfaces is satisfied).

## 3.11 Meta-Programming using builder specification

The Think compiler provides a way to specify, for each component, which builders should be invoked. This is done using the `builder` property. TO BE COMPLETED...

## 3.12 Entering component world

Before components can be called, a booting system must enter the ”component world”. This means that at boot time, there is no notion of component and the system will want at a given time call its first component instance. This case also arises when an interrupt handler is made out of component. Nuptseprovide several way to ”enter” a component instance. The simplest way is when the first component to call is single, or more precisely when its code is not shared with other instances. In this particular case, the default builders allow architecture artifact of the instance to be accessed from normal C code because in the generated code there is no ambiguity on the identifier of the component instance. For example a client method can be called from a C function even though the function is not a private or server method.

Another method is to use the `THIS` and `SET_THIS` special keywords. The first one returns the identifier of the component instance and can be used when there is no ambiguity on the component instance. The second one indicates to the compiler a runtime expression that represents the identifier of the component (a priori a variable initialized with `THIS`): it can be used to set the identifier if a component in a C function, or switch to another component instance.

The above facilities can be used in the following typical cases:

- system boot;
- interrupt handlers;
- component wrapping of legacy code.

They should not be used in the general case of single components where C functions can be easily declared as private methods, because they make assumptions on the ability of the backend to handle these facilities and because the component may be later turned into a non-single component.