Think User's Manual

1 Introduction

The goal of this document is to give sufficient information to help users of the Think platform to design and build component-based software. tool chain and languages a straight and simple use of the Nuptse version of the Think framework.

Think is a native implementation of the Fractal component model[?]. Think (ThinkIs Not a Kernel) can be used to develop OS kernels but Think is by no means specialized to this domain (the framework can be used to develop any C system or application).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 pre-defined software component and rapid porting of execution infrastructure on new hardware targets. The current Think release hosts a build chain and several language specifications.

Since its first design, Think has known several transformation throuh several version. The latest, called Nuptse focuses on simplcity and efficiency:

- it simplifies the burden of the developpers of think-based software by providinga simplified languages, espacially for developpers 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.

For this reasons, 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 artefacts.

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

This document is structured as follows. Section 1.1 gives some imputs on the general organisation of the sources of the projet, section 2 explains how to download and install the build chain and section 4 shows how to compile a software through a helloworld example. Section 5 then details the key concepts required to design a Think-based software. The document finally describes the different languages of the platform: theInterface Description Language (IDL) in section 7, the Architecture Description Language (ADL) in section 8 and the Component Programing Language (CPL) in section 9.

1.1 Organization

The Think project is hosted on the Objectweb server (http://think.objectweb.org). The URL of the subversion repository is: svn+ssh://svn.forge.objectweb.org/svnroot/think. The repository is structured ad follows:

- trunk : trunk of the v2 version of the framework.
- branches : branches of the v2 version of the framework.
- V3 : repository of the v3 version of the project (trunk and branches)
- Nuptse : repository of the 'nuptse' version of the framework
- Codegen : source code of a package oriented for code generation and used in the nuptse version

This document focusses on the nuptse version. The trunk and branches are structured as follows:

- fractal-c: includes the fractal idl files. All the control interfaces defined by the fractal model (CI, CC, BC, LCC) are defined there.
- think: implementation of fractal interface in c
- thinkadl: contains the build chain which is the core of the Think project. This directory includes all the tooling that is needed for code generation and for driving the gcc based compilation process of the generated files.
- kortex: contains an OS-oriented component library
- tools: this directory includes build tooling, scripts....

2 Installation

The build chain can be retrieved from the ObjectWeb site using svn.

2.1 Getting sources

Type 'svn co svn+ssh://developername@svn.forge.objectweb.org/svnroot/think/nuptse/trunk'. Type 'cd trunk/thinkadl; ant dist' to compile the build chain.

2.2 Getting pre-compiled binaries

Type 'svn co svn+ssh://developername@svn.forge.objectweb.org/svnroot/think/nuptse/releases'. Type 'cd releases; tar xvfz thinkadl *release number*.tgz', where *release number* refers to the release you want to use.

3 Requirements

Think Cross Tool chain can be deployed on many platforms. The required tool set on the targeted platform can be checked on the list below:

- ant (preferably the last version... at least the 1.7.0 that can be downloaded on http://ant.apache.org/
- java 5
- a C compiler

4 Quick Start: helloworld example

The build chain takes as input a (Fractal Compliant) architectural description of the tar-geted kernel written in ADL/IDL and a repository of component implementations ADL (Architecture Description Language) and IDL (Interface description language) are usedto describe the included components instances and the way they are interconnected in order to build up the kernel.

5 Key concepts

5.1 Properties

6 How to compile an architecture

7 The Interface Description Language (IDL)

8 The Architecture Description Language (ADL)

8.1 Notations

8.2 Keywords

8.2.1 component

Usage Declares a component definition.

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

Description Declares a component definition (i.e. a component type) named `compDefName`. This definition may extends another definition named `extCompDefName`. Extending a component definition is like inlining the whole content of the extended definition into the extending one. Abstract component definition are component definition for which the content is not totaly defined. Note that only abstract component declarations (see **??** and 8.2.6 for more details) can be of an abstract component definition.

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 = sumCompX
  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
}
```

Above definition 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
}
```

8.2.2 provides

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

```
provides <itfType:DotName> as <itfName:Name>
        [ in <implemName> ]
```

Description Declares a provided (a.k.a server) interface named `itfName` of interface type `itfType`. ****** TODO:single **** **** TODO:implem name **** **** TODO:single **** **** TODO:in implem ******

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 foo as here.is.Foo
}
```

8.2.3 requires

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

```
requires <itfType:DotName> as <itfName:Name>
        [ ( mandatory | optional ) ]
        [ in <implemName> ]
```

Description Declares a provided (a.k.a server) 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. ****** TODO:in implem ******.

Example 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 foo as here.is.Foo
}
```

8.2.4 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 speficied 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 instanciated 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
}
```

8.2.5 assigns

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

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

Description Assigns value `value` to attribure `attName` of sub-component `subCompName`. `subCompName` must be the name of a sub-component declared in the component definition (see 8.2.6). 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
}
```

8.2.6 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 a abstract sub-component and must be used If and only if `compDef` is an abstract component definition. In that case, the enclosing defintion must also be 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` of abstract component type `here.is.foo1`, and a (concrete) component definition `here.is.bar1` containing a (concrete) sub-component `c` of (concrete) component type `here.is.foo1`.

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

component here.is.foo2 {
        ...
}

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

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

8.2.7 singleton

Usage Forces a component definition to be instanciated only onced in a architecture..

```
singleton
```

Description Forces a component definition to be instanciated only onced in a architecture. Two component definitions that contain a declaration of a sub-component of such a singleton definition will share the same instance at runtime.

Example The following code declares a singleton component definition `here.is.foo`, a component definitions `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 type `here.is.bar2`, `x:c2` and `x:c3:c1` are aliases and represent the same 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
}
```

8.2.8 content

Usage Specifies a file that contains implementation code.

```
content <fileName:DotName>
        ( [ for <impName:Name> ]   | [ raw ] )
```

Description Specifies that file which base name (i.e. without extension) is `fileName` with dot replaced with file separator, contains implementation code for implementation `impName`. 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. If `impName` is omited, then the file will be interpreted as containing code for the default implementation (see **??**). Note that

code of a particular implementation may be spread across several file.
If **`raw`** is specified, then the file is to contain code that does not directly implement server interfaces but normal code instead and cannot make use of the model artefacts (access attributes, call client interfaces, ...).
Files will be searched in the component reposiroty path list specified in the command (see **??**).

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, files `rep1/a/b/c/f2.c` and `rep2/a/b/f3.c` contain code for the implementation `imp1`, file `rep2/a/b/c/f4.c` contains code for the implementation `imp2`, and file `rep1/a/b/d/f4.c`is to be added as is. Note that `rep2/a/b/c/f2.c` will be ignored.

```
component here.is.foo {
        ...
        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
         +- d
             +- f5.c
```

8.2.9 cflags

8.3 Deprecated Keywords and constructions

9 The NuptC Component Programming Language

Nuptse provides a Component Programming Language (CPL) called NuptC to develop functional code implementing the provided interfaces of component. Contrary to previous CPLs of Think, NuptC has been defined in order to:

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

The CPL extends the C language with architectural-oriented keywords.This is achieved without extending the C syntax but instead providing keywords through reserved identifiers having well defined naming conventions. In order to allows arbitrary implementations and optimisations of the glue, thes identifiers do not reflect any particular meta-data organization. These identifiers reify the architectural definitions found in the corresponding ADL file, so that a programmer can declare a C function as implementing a method of a declared server interface, call a method of a declared client interface, access a declared attribute, etc.

Functional code is parsed by the chain and is translated into an Abstract Syntax Tree using the Codegen library[2] (because nuptse does not extend the C grammar, files can be parsed with the C parser provided by codegen and define a listener to handle the specific keywords). This AST 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). This section details how to develop functional code using the Nuptse CPL.

9.1 Declaring a method of a server interface

The implementation of a method `foo` of a server interface `bar` can be declared as follows[3]

```
int SRV_bar__foo(int a, char b) {
  ...
}
```

[2]Codegen is currently hosted by the project.

[3]Note the double underscore (__) between the interface name and the method name.

9.2 Declaring private variables

Components can have a private variable, that will be instanciated once for each component instance. A private variable is very similiar to an atribute in its usage but differs in the fact that it is not declared in the architecture description and so is not visible nor modifiable from outside the component. This variable must be named `PRIVATE` and can be of arbitrary type, thus, using the `struct` construction of the C language, allowing the declaration of virtually any number of private data. For example :

```
int PRIVATE ;
```

or

```
struct {
  int i ;
  int j ;
} PRIVATE ;
```

This variable can be referenced in the code just like any other variable.

9.3 Calling a client interface

A client method `foo` of a client interface `bar` can be invoked as follows :

```
SRV_itf__meth(...) {
  ...
  CLT_bar__foo (0, 1);
  ...
}
```

Note that only server and private methods can call client methods.

9.4 Referencing an attribute

An attribute `foo` can be referenced as follows :

```
int i = ATT_foo;
ATT_foo = 2;
ATT_foo++;
```

Note that constant attributes can not be modified, so the last two examples will no be allowed in this case.

9.5 Declaring a private method

A private method is a C function that make use of architectural artifacts. A private method foo can be declared as follows:

```
PRV_foo(...) {

}
```

Only server methods and private methods can access attributes or call client interfaces.

9.6 Advanced keywords for controller programming

NuptC also provide keywords for programming implementation of Fractal 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 `META_`.

9.6.1 Client Interfaces

`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 if the table is transformed into a 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);
```

9.6.2 Server Interfaces

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 if the table is transformed into a 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.

9.6.3 Attributes

META_NB_ATTS Compile-time value representing the number of client interfaces 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);
```

9.6.4 Components

`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 if the table is transformed into a 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.

10 Extension Mechanism

The Architecture Description Language introduced in section 8 allows to define a component by extending another definition. Starting from a initial definition it is possible to add new interfaces, new attributes or nes 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. The extension mechanism give a way to extend multiple component definitions in a generic way using pattern matching. An extension specification is a standard ADL file but where names can have "jokers" that may match 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]
}
```

The build chain takes a option ”ext-files” with ”:” separated For each component specification found in the architecture given as input to the build chain, the extension definition is matched to see