The ObjectWeb Consortium

THINK Cookbook

Authors:
M. Leclercq (STMicroelectronics)
A.E. Özcan (STMicroelectronics)

Released June 8, 2006
Status Draft
Version 0.1

Contents

1 Introduction

Component-based software engineering (CBSE) appeared in the ten past years as a novel development methodology to keep pace with the increasing complexity of software infrastructures such as operating systems and middleware. Although the component based approach is a defacto methodology in many other engineering domains, the intrinsics of the software makes difficult its adoption , and the CBSE remains currently a research subject [References].

If there exists many definitions for software components, their common features can be summarized in a few set of concepts. A key issue of CBSE is isolation in terms of code and instance data. The isolation is insured thanks to interfaces where the only access points of components are their interfaces. In other words, components receives invocations via their (server) interfaces and emits invocations via their (client) interfaces. Second, components need to be connected to enable their interactions. Connections are done between interfaces and are unidirectional in the most of the component models.

These limited set of constraints clarify the software architecture, at least more than objects, since the borders of components are clearly identified with their interfaces and their relationship is captured thanks to explicit connectors. Consequently, many tool-chains, including THINK , provide support for component assemblies using architecture description languages (ADL), and generate the glue code integrating the components to build the specified software. This offer an invaluable help to software development since it allows for some statical analysis such as assemble correctness verifications, and relieve programmers from often tedious and error prone configurations tasks.

1.1 Purpose of this document

This document is an introduction to THINK. It explains informally what do mean some keywords that are commonly used such as FRACTAL, ADL, IDL and so on. It presents the structure of the THINK development tool-chain and explains how it is used. In particular, this document illustrates two basic examples to help new programmers to start programming with THINK components.

This document does not present the FRACTAL component model nor component-based programming in general. We invite readers who are interested in these subjects to take a look at the references given at the end of this document.

1.2 FRACTAL and THINK

1.2.1 FRACTAL

FRACTAL is component model which is inline with this issue. FRACTAL is a reflective component model which can be specialized to satisfy specific needs and can be implemented in different programming languages from C to Java to target different platforms. Its original features beside other component models are :

Recursivity : a FRACTAL component can either be implemented in a general purpose programming language or be composed by other components. This let architects to have a uniform view of applications at various levels of abstractions.

Component sharing : In FRACTAL , a given component instance can be included (or shared) by more then one components. This feature us very useful to model shared resources such as memory managers or device drivers.

Binding components : FRACTAL offers a single abstraction for component connections that is called bindings. Bindings can embed any communication semantics from synchronous method calls to remote-procedure calls.

Reflectivity : FRACTAL components are optionally reflective in the sense that they can be monitored and controlled.

Execution model independence : FRACTAL does not impose nor specify any execution model. In that, FRACTAL components can be run within other execution models than the classical thread-based model such as event-based models and so on.

1.2.2 THINK

THINK is a development environment for programming FRACTAL components in C and C++ languages. This environment includes a compiler that for building software systems from their architecture descriptions. In other terms, the role of this compiler is to read a set of architecture description files defining the configuration of a given software and to build the corresponding software by gathering the components to be used from component libraries and generating the glue code to fit them in the required configuration, and also to compile the set of generated and hand-written implementation files to produce an executable application, or a bootable kernel. Thanks to this compiler, primitive THINK components are programmed with respect to a programming guide on top of C and C++ programming languages to allow for expressing component-related aspects such as instance data, interface implementations, invocations and so on. Note the generated code conforms to standard C, and the compiler used to compile both implementation and generated files is a standard C or C++ compiler and linker for a given target hardware platform.

THINK environment is designed to provide component libraries to allow system architects for building software by reusing on-the-shelf components. Currently, only a single library is provided for building operating system kernel instances but our current work aims at extending this tool-chain with many other libraries for building middleware and multimedia systems.

1.2.3 Definition of terms

ADL Architecture description languages are in general high-level declarative languages allowing architects to describe software configurations in terms of components, interfaces, connections between components and component compositions. They are mostly supported by some statical correctness analysis and code generation tools.

IDL Interface description languages are specific languages with simple syntax for describing component interfaces. Their primary role is to have a common language between different software components. They can be used for type based compatibility analysis and for generating communication adaptors for component couples that can not be directly connected.

PPL Primitive programming language refers to a general purpose programming language such as C, C++ or Java which is used to implement the behavior of primitive software components.

ADL Compiler An ADL compiler is generally in charge of reading a software architecture (ADL and IDL) and the implementation of referred components (in PPL) to generate the corresponding software.

2 Getting started with THINK development tool-chain

2.1 Getting the sources

Sources of the THINK development tool-chain are available on the ObjectWeb subversion reposiroty. One can checkout the current revision of the THINK project using the following command:

```
svn checkout svn://svn.forge.objectweb.org/svnroot/think/v3/trunk
```

THINK sources are organized in different sub-project. Each sub-project contains its own source tree.

fractal-c contains the FRACTAL controller's API written with the THINK IDL language.

think contains the C implementation of FRACTAL controllers (component-identity, binding-controller, lifecycle-controller, ...)

kortex is the component librairy dedicated to the construction of operating system kernel instances.

thinkadl contains the compiler that reads architecture descriptions to produce compiled applications.

tools contains ANT scripts for building or synchronizing above sub-projects.

Some of the above sub-projects are not standalone but can be used independantly by other sub-projects. Such sub-projects contain a folder namely `externals` folder to include archives of the sub-projects on which it depends. For instance, the *kortex* sub-project contains, in its `externals` directory, a jar file of sources of *fractal-c* and *think* sub-projects. The *tools* sub-project contains ANT scripts that ease the synchronization of sub-projects (the description of these scripts is beyond the scope of this document).

2.2 Installation

To develop a new THINK application, you first need to compile and install the THINK ADL compiler and install default libraries (*fractal-c*, *think* and *kortex*). This can be done using the ANT script of the *tools* sub-project.

To install the THINK development tool-chain goes in the `tools` directory and run:

```
ant install -Dprefix=PREFIX
```

Where `PREFIX` is the path where you want to install the tool-chain.

The install directory contains the following files and directories:

arch directory contains files defining target architecture specific options such as default C-Flags, C compiler command, etc.

execute.xml and execute.property are used to run the THINK ADL compiler.

externals directory contains Java libraries used by the THINK ADL compiler.

lib directory contains the default libraries that can be used to build THINK application. This includes `think.jar`, `fractal-c.jar` and `kortex.jar`

template directory contains template files that can be used to create a new THINK application. See section 2.3 for more detail.

think-adl.jar contains the Java classes of the THINK ADL compiler.

2.3 Using the THINK ADL compiler

The THINK ADL compiler reads an architecture description, generates corresponding C source code and compile the resulting application. Since the THINK ADL compiler drives all the compilation process of the application, it provides many command line options. To ease its the execution, we provides template ANT files that can be used and tuned to meet your needs. These files are located in the `template` directory of the installation directory. To use them you simply have to copy them in your working directory.

build.xml is an ANT script that defines a *compile* target which executes the THINK ADL compiler.

compiler.properties defines properties used by the `build.xml` ANT script to locate the THINK ADL compiler and the standard librairies. You should not modify this file.

build.properties defines compilation parameters like the name of the ADL to compile, the target architecture, the path where the compiler may find your sources, etc. You must modifiy this file, at least to specify the name of the ADL to compile.

When the `build.properties` file is configured correctly, you can compile your application by using the `ant` command.

2.3.1 ADL, IDL, source file names and `src-path`

The THINK ADL compiler uses a Java-like naming convention to find files (ADL, IDL and source files).

Let's say that you decide to put all your source files in a `src` directory. First you have to specify this `src-path` in the `build.property` file. Therefore, every file in this directory are named by the compiler according to their relative path, for instance, a file who's path is "`src/dir1/dir2/MyInterface.idl`" is named by the compiler "`dir1.dir2.MyInterface`" and can be referred in an ADL file using this name. [1]

[1] More precisely, we can say that this file contains a declaration of the interface "`MyInterface`" in the *package* "`dir1.dir2`", and its *fully qualified name* is "`dir1.dir2.MyInterface`".

The THINK ADL compiler allows for specifying multiple `src-path` as a set, which means that the compiler will search files in different base directories. For instance if the `src-path` is "`src1:src2`", then if the compiler must find an interface who's name is "`fractal.api.ComponentIdentity`", it will try to find a file who's name is "`src1/fractal/api/ComponentIdentity.idl`" or "`src2/fractal/api/ComponentIdentity.idl`".

Moreover, you can specify archives with a zip file (or a jar file) in the `src-path`. In this case, the compiler will decompress the file in a temporary directory, and will use this directory as a normal source directory.

2.3.2 Files produced by the THINK ADL compiler

The THINK ADL compiler produces many files (source code, object file, etc.). The path where these files are placed is specified using the `out-path` option in the `build.property` file.

There are three different types of generated files

- The source codes generated from ADL files are put in the "`adl`" directory. For each component that has been defined in the compiled architecture, a "`.adl.c`" file is generated. This C source file contains the declaration of the component internal data structures (server interfaces, client interfaces, private data, etc.), the definition of component specific macros that can be used in the implementation files and in the case of static instantiations, the instance definitions of components.
- The files that are generated from IDL definitions are put in the "`idl`" directory. For each interface used in the compiled architecture, at least one "`.idl.h`" file is generated. This header file contains the translation of the interface declared in the IDL language into C function prototypes and virtual function table structures.
- compiled object files and the executable files are placed in the "`obj-`*arch*" directory.

It may be usefull to have a look on these generated files, especially for debug purposes. In particular, it is sometime useful to read the "`.idl.h`" files generated for a given interface to see how a method declared in the IDL language is translated into a C function prototype, to declare the same in your implementation files.

2.4 The Kortex component library

The Kortex component library contains a set of components and interface definitions dedicated to the construction of operating system kernel instances.

The sources of the kortex sub-project are separated in four different directories:

- `src/generic` contains target independant sources. The components found in this folder can directly be reused for different hardware platforms. This folder also contains some component type definitions and IDL files for platform dependent components that are may be implemented in the `src/arch` folder.

- `src/arch` contains architecture dependant sources such as trap handlers implementation, context switch, etc.
- `src/chip` contains chip specific sources.
- `src/platform` contains platform specific sources.

Note that all these directories are considered as base paths (i.e. different "`src-path`" elements). Therefore, for instance, the interface defined in the file `src/generic/activity/api/Main.idl` is named `activity.api.Main`.

The `src/generic` source directory contains (among others) the following packages :

activity component related to thread creation and scheduling. It also contains components for semaphore mechanism and lock prevention.

boot generic bootstrap components and interfaces.

input two types of input devices are specified in this directory, the touch screen and the pen.

irq this directory contains generic interrupt management components and interface definitions.

libc source code of a partial implementation of classic libc.

loader dynamic loading support components for binary files in memory.

log some logging support components.

memory generic memory management components. These components are based on the HAL that whould be ported for the target platform. Generally, components implementing the memory HAL provides "Sbrk" interface for obtain underlying memory. "Sbrk" is used to increment the program's data space and second to offer transparency to the underlying address space model use by the personality.

net network stack components that are implemented following the x-kernel philosopy.

video components implementing graphical and textual display services.

3 FRACTAL ADL

The THINK framework supports code generation from architecture descriptions that are written in FRACTAL ADL. This section presents how component architectures can be described in FRACTAL ADL through a basic *helloworld* example. It also gives the FRACTAL ADL specification describing the language grammar. For a more detailed introduction to FRACTAL ADL language we invite readers to see the FRACTAL ADL tutorial .

The FRACTAL ADL is a declarative language allowing programmers to describe software configurations in terms of interfaces, attributes, component compositions and bindings. The FRACTAL ADL is syntactically based on XML. The extension used for FRACTAL ADL files is `.fractal`. The FRACTAL ADL is caption sensitive language. Each ADL file contains a single component definition whose name is equal to the file name.

Following the recursive composition rational of FRACTAL , any software architecture is described in a top level component; which is generally composed by other components. That is, the entry point of any FRACTAL software is defined by one top level component that is described in a given `.fractal` file that references optionally other `.fractal` files.

3.1 Basic ADL example

An example of basic *helloworld* description is depicted in Figure 1. In this example, the top level of the described software is the `HelloWorld` component, providing the `activity.api.Main` interfaces that gives access to its entry point. It contains two subcomponents, respectively the `client` and the `server`. The definition of the `client` component is given in a separate `.fractal` file while the definition of the `server` component is inlined. Finally, the `HelloWorld` component defines two bindings: the first one connects its `main` interface to the one provided by the `client` component, the second one satisfies the `printer` interface of the `client` component by connecting it to the server component.

```
1  <!-- file = ClientType.fractal -->
   <definition name="ClientType">
3   <interface name="main" role="server"
              signature="activity.api.Main"/>
5   <interface name="printer" role="client"
              signature="printer.api.Service"/>
7  </definition>

9  <!-- file = ClientImpl.fractal -->
   <definition name="ClientImpl" extends="ClientType">
11  <content class="ClientImpl"/>
    <controller desc="primitive"/>
13 </definition>

15 <!-- file = HelloWorld.fractal -->
   <definition name="HelloWorld">
17  <interface name="main" role="server"
              signature="activity.api.Main"/>
19  <component name="client" definition="ClientImpl"/>
    <component name="server">
21   <interface name="printer" role="server"
               signature="Service"/>
23   <content class="ServerImpl"/>
     <controller desc="primitive"/>
25  </component>
    <binding client="this.main" server="client.main"/>
27  <binding client="client.printer" server="server.printer"/>
   </definition>
```

Figure 1: ADL description of a HelloWorld application that is composed by a client and a server.

3.2 FRACTAL ADL specification

The FRACTAL ADL has a very simple structure and a straightforward grammar. Tables 1 to 14 presents the grammar of each construct of this language. Note that, a *sub-element* of a tag corresponds concretely in FRACTAL ADL to an XML tag that can be accepted as a child of the given tag;

and the fields of a tag corresponds to the arguments that can be passed in the tag. The mandatory fields are denoted with `verbatim` characters while the optional ones are denoted with *italic* characters.

Each description is written in a base XML module, namely *definition* (Table 1), which encapsulates a set of sub-elements. In the case of primitive component definitions, this base module may contain a set of *client* and *server interfaces* (Table 3), some *attributes* (Tables 5 and 6). Primitive components are implemented in a PPL, so that its implementation file is specified with a *content* element (Table 8). The content element accept a *directive* element to specify compilation directives (Table 10). Finally, a *controller* description (Table 7) may be put to specify the controller interface implementation generator for a component.

In the case of composite component definitions, the *content tag* makes no sense since the component in composed by other components rather then be implemented in a primitive programming language. Hence, it contains a set of *component* (Table 2) and *binding* (Table 4) elements.

FRACTAL ADL supports modular descriptions through two forms of inheritance. First, a definition can extend another one by defining additional/overloading features (i.e. *ClientImpl* extends the *ClientType*). Second, *component tags* defining a sub-component can refer to a *definition* written in another XML file (i.e. The *client* sub-component of *HelloWorld* refers to *ClientImpl*). Note that this tag may be used to add/overload features, simply by considering that the referenced definition is an inherited base component.

`definition` : Component architecture definition		
Sub-elements : comment*,interface*,component*,binding*,content?, attributes?,controller?,template-controller?,logger?, virtual-node?,coordinates*,output?		
Fields :	`name` *arguments* *extends*	The name of the component definition. Arguments that customize this definition. Specifies the name of the extended component definition.

Table 1: Component definition tag.

`component` : Component instance definition		
Sub-elements : comment*,interface*,component*,binding*,content?, attributes?,controller?,template-controller?,logger?, virtual-node?,coordinates*,output?		
Fields :	`name` *definition*	The name of the component instance. Reference to a component definition.

Table 2: Component instance tag.

`interface` : Component interface definition		
Sub-elements : comment*		
Fields :	`name`	The name of the interface
	`role`	The role of the interface [`client` | `server`]
	`signature`	The signature name of the interface
	contingency	Indicates if the interface is mandatory or optional [`mandatory` | `optional`]. Default value is `mandatory`.
	cardinality	Indicates whether the interface accepts multiple connections or not [`singleton` | `collection`]. Default value os `singleton`.

Table 3: Interface definition tag.

`binding` : Binding definition		
Sub-elements : comment*		
Fields :	`client`	The client end of the binding `componentinterface`
	`server`	The client end of the binding `componentinterface`

Table 4: Binding definition tag

`attributes` : Component attribute set definition. This tag can contain multiple `attribute` elements for value assignments to attribute fields.		
Sub-elements : comment*,attribute*		
Fields :	`signature`	The signature name of the attribute structure.

Table 5: Attribute set definition tag.

`attribute` : Attribute assignment definition.		
Sub-elements : comment*		
Fields :	`name`	The name of the attribute field.
	`value`	The value to be assigned to the field.

Table 6: Attribute field assignment tag.

`controller` : Controller generator specification. If this tag is not used in a component definition, a default controller generator is used by the ADL compiler.		
Sub-elements : comment*		
Fields :	`desc` *language*	The descriptor of the controller generator. The language in which the controller interfaces must be implemented by the generator. By default, the language value is the same as the content implementation language.

Table 7: Controller specification tag.

`content` : Primitive component implementation file specification.		
Sub-elements : comment*,directive*		

Fields :	`class`
	
	`language`
		The implementation file name without its language indicator extension.
		Specifies the language of the implementation file. Currently supported languages are `thinkC`, `macroC` and `cpp`.

Table 8: Content specification tag.

`comment` : A tag for including commentaries in ADL descriptions.		
Fields :	*language* `text`	Specifies the language of the comment. Comment text.

Table 9: Commentary tag.

`directive` : Specification of compilation directives for primitive components. This tag accept many compiler and linker flags as sub elements. It also accepts a set of `include` tags specifying files to be included.		
Sub-elements : include*,cflag*,ldflag*		
Fields :	`language`	Indicates for which language these compilation directives are.

Table 10: Compiler directive specification tag.

`cflag` : A compiler flag to be used.		
Fields :	`value`	A text indicating a compiler flag

Table 11: Compiler flag specification tag.

`ldflag` : A linker flag to be used		
Fields :	`value`	A text indicating a compiler flag

Table 12: Linker flag specification flag.

`include` : Additional implementation files to be included.		
Fields :	`file`	A file name

Table 13: Implementation file inclusion flag.

`output` : Output format of the compilation result		
Fields :	`format`	The output format extension.

Table 14: Compilation output format specification tag.

4 THINK IDL

THINK specifies a strongly typed IDL. Interface descriptions are used by the compilation tool-chain to make type based analysis to verify the compatibility of interconnected component interfaces, and to generate communication adaptors for component pairs that can not be naturally bound (JNI, RPC, etc.).

4.1 Basic example

THINK IDL supports two types of constructs to describe interfaces and records that can be used for complex data structure definitions. Records are also used for defining component attribute structure definitions.

Each interface or record definition must be done in a separate file. The extension used for these files is /tt .idl. The file name must correspond to the interface or record name which is defined in the file. Each IDL definition belongs to a package. The package name must correspond to the path in the file system.

An IDL definition can reference a type which is defined in another IDL definition. To this end, two kinds of identifiers may be used, a simple identifier or a qualified identifier. A qualified identifier corresponds to `<package>.<interface>` giving the absolute description of the IDL definition. An identifier corresponds only to `<interface>`, and may be considered as a relative path. An single identifier can be used to reference an IDL type which is defined in the same package. Otherwise, one may import the qualified identifier to use then an identifier in the IDL definition. Note that the imported identifiers overload the identifiers found in the same package.

The below IDL definition specifies an interface, called `Printer`, in the package `video.api`, which provides two distinct methods, one providing a print operation for a character, and the other for a string.

```
//file = video/api/Printer.idl
package video.api;

public interface Printer {
   void printChar(char c);
   void printString (string str);
}
```

The below IDL definition defines a record called `Service attributes` in the package `attributes` which have two integer fields respectively called `cols` and `rows`.

```
// file = attributes/ServiceAttributes.idl
package attributes ;

record ServiceAttributes {
  int cols ;
  int rows ;
}
```

4.2 THINK IDL specification

This subsection presents the grammatical specification of the THINK IDL language. We present this grammar in three parts which are the common elements between both record and interface constructs

and their specific elements.

4.3 Common grammar elements

The very first element of the IDL language consists in the identifiers. As explained above, two kinds of identifiers are supported : qualified identifier and the simple identifier. The simple identifier corresponds to a string corresponding to a classical C identifier (starting with an alphabetical character and not containing arithmetical or boolean operator characters). A qualified identifier is a list of identifiers, connected with points.

```
QualifiedIndentifer ::= ( Identifier ( "."Identifier )*
```

An IDL declaration starts with a package clause, followed by a list of import clauses, and contain one interface definition or one record definition.

```
Declaration ::= Package (Import)* (ItfDecl | RecDecl)
Package     ::= "package" QualifiedIdentifer ?;?
Import      ::= "import" QualifiedIdentifer ?;?
```

Another common element is the supported types. We distinguish between primitive types corresponding to keywords defined in the IDL language, and the complex types that are defined in other IDL definitions.

```
Type ::= PrimitiveType | ComplexeType
ComplexeType ::= QualifiedIndentifer | "any"
PrimitiveType ::= (
"boolean"  |
"char"     |
"byte"     |
"short"    |
"int"      |
"long"     |
"float"     |
"double"   |
"string")
```

Primitive types are mapped to C types when they are compiled. Table 15 presents the mappings that of each primitive type with their shortcuts that can be used in the implementation files.

A C identifier is generated for complex types. The identifier is generated respected the rule given below.

```
( Identifier ( "."Identifier )* => R( Identifier ( "_"Identifier )*
```

4.4 Interface definition grammar

An interface type declaration identified by Identifier contains two sorts of declarations; methods and constant fields. The keyword public isn?t relevant and is only here for compatibility purpose.

```
InterfaceDeclaration ::=  "public"? "interface" Identifier "{"
(
 MethodDeclaration |
 FieldDeclaration |
 ";")*
"}"
```

IDL type	C type	Shortcut
void	void	
boolean	unsigned char	jboolean
char	unsigned short	jchar
byte	signed char	jbyte
short	signed short	jshort
int	signed int	jint
long	signed long long	jlong
float	float	jfloat
double	double	jdouble
string	char *	
any	void*	

Table 15: Mappings for primitive types and their shortcuts.

The grammar of a method declaration is given below. The keyword "..." is used for passing arbitrary parameters number.

```
MethodDeclaration ::= ResultType Identifier FormalParameters ";"

ResultType ::= Type | "void"

FormalParameters ::=
 "("
 FormalParameter
 ( "," FormalParameter )*
 ( "," "..." ) ?
 ")"

FormalParameter ::= Type Identifier
```

A constant field declaration grammar looks like this. The expression must be a valid constant expression and only a primitive type can be assigned as a constant. No semantic check on the expression is done by the IDL compiler.

```
FieldDeclaration ::= PrimitiveType Identifier = Expression ";"
```

4.5 Record definition grammar

The grammar of the record definition is given below. A record can contain any types. Note that, a record can not contain another record, so that references to other record identifiers are translated as pointers.

```
InterfaceDeclaration ::=   "record" Identifier "{"
  (BasicFieldDeclaration)*
"}"

BasicFieldDeclaration ::= Type Identifier ";"
```

5 THINK primitive programming languages

THINK supports different programming languages for implementing primitive components, each having different features compromising the programming easiness to the control of the programmer on the final implementation code. Among these languages, the *thinkC* language defines some rules and conventions to respects to program with components in C. It is the most transparent one where the programmer is in charge of defining and using all the data structures related to the components. The *thinkMC* provides some macro-based constructs to provide an abstraction of mentioned conventions and henceforth to ease the programming of components. It improves drastically the lines of codes in the implementation files. Finally, C++ is also supported for programming components. This language is definitely the simplest one for implementing components since it provides intrinsic constructs for dealing with instances but may have the drawbacks of using C++ instead of C.

Note that components written in those three languages are completely interoperable, so that one can choose an implementation language depending on some specific needs for a given application, and consequently compose a heterogeneous component library.

5.1 thinkC

TODO

5.2 thinkMC

The purpose of thinkMC is to abstract the implementation programmer from the conventions and the definition of data structures related to components. Following subsections give respectively a basic example of component implementation in thinkMC and the specification of constructs of this language.

5.2.1 Basic example

In this subsection, we explain how primitive components may be programmed in thinkMC through two basic implementation examples. The code depicted in Figure 2 may be the implementation of the `client` component that has been presented in Figure 1. The code starts by a `DECLARE_DATA` construct that must be used to declare the instance data of the component. Note that this construct must be present in every *thinkMC* file, even if the component does not have any instance data. This construct must be followed by the inclusion of the header file called `think.h` that is in charge of defining the macros that may be used by the programmer to express component related issues.

Recall that this component provides an interface called `main` and requires an interface `printer`. The implementation of the interface `main` starts at line 5 of the code. This interface contains only one method which is itself called `main`. Note that each method implementation must accept as the first parameter a pointer called `_this` that gives access to the instance data of the component (i.e. in C++). The rest of the parameteers aare the ones that are declared in the IDL definition, respecting the shortcuts defined in Table 15. This method emits some invocations on its client interface, called printer). It access to this client interface thanks to `REQUIRED` keyword, and emit the invocation using the `CALL` keyword.

```
1  // file client.c
2  DECLARE_DATA {};
3  #include <think.h>
4
5  void METHOD(main, main)(void *_this, int argc, char** args) {
6    CALL(REQUIRED.printer, printChar, '\n');
7    CALL(REQUIRED.printer, printString, "Helloworld");
8    CALL(REQUIRED.printer, printChar, '\n');
9  }
```

Figure 2: Implementation of a client component using a printer service to display messages.

The implementation of the `server` component is quite similar to the one of the `client` component. This component has two integer instance variables, respectively called `x` and `y`. It implements two interfaces: the first one is the `printer` that is used by the client components, and the implementation of this interface consists in the implementation of both `printChar` and `printString` methods as defined in Section 4. The second interface is indeed a control interface that is in charge of handling with the life cycle of the component as described in the FRACTAL specification. The `start` method of this interface is invoked by the execution environment before the component becomes functional. It is traditionally used to initialize the instance variables of the components, and can be considered as a constructor. Similarly, the `stop` method is called when the component will be destroyed and can be considered as a destructor.

```
1  // file server.c
2  DECLARE_DATA {
3    jint          x, y;          /* Cursor position */
4  };
5  #include <think.h>
6
7  void METHOD(printer, printChar)(void *_this, char c) {
8    printf("[%d, %d] %c", DATA.x++, DATA.y++, c);
9  }
10
11 void METHOD(printer, printString)(void *_this, char *s) {
12   printf("[%d, %d] %s", DATA.x++, DATA.y++, s);
13 }
14
15 void METHOD(lifecycle_controller, start)(void *_this) {
16   DATA.X = 0;
17   DATA.y = 0;
18 }
19
20 void METHOD(lifecycle_controller, stop)(void *_this) {
21   DATA.X = 0xDEAD;
22   DATA.y = 0xDEAD;
23 }
```

Figure 3: Implementation of a printer server component.

5.2.2 thinkMC constructs specification

Implementation file The implementation file must be called with the name that haas been used in the specification of the component in ADL. The extension of the file must be ċ. The content of the file must follow the below structure.

```
IMPLEMENTATION ::= DECLAREDATA "#include␣<think.h>" (METHOD)*
```

Instance data declaration The declaration of instance data of components is made using the `DECLARE_DATA` construct whose structure is given below. This declaration must be done even if the component does not have any instance data. Instance data of a component is formed by a set of fields.

```
DECLAREDATA ::= "DECLARE_DATA␣{" (fieldDeclaration)* "}␣;"
```

Method implementation A component implementation file must contain the implementations of each method of each interface that is provided by the component. There are two issues to consider for method implementations : declaring the signature of the method with a specific construct called `METHOD`, and accept as first parameter of the method, a void pointer which is called `_this`. The structure of the method implementation is given below.

```
METHOD ::= returnType "METHOD(" interfaceName "," methodName ")"
                              "(void␣*_this" ("," argument)* ")"
           "{"
             ImplementationCode.
           "}"
```

Method invocation Components can invoke methods that are provided by other components through their client interfaces. To this end, a `CALL` construct has been defined whose structure is given below.

```
INVOCATION ::= "CALL␣(" interfaceName "," methodName ("," parameter)* ");"
```

Access to client interfaces Components can access to their client interfaces using the `REQUIRED` construct. The structure of this construct is given below.

```
ACCESSITF ::= "REQUIRED." interfaceName
```

Access to instance data Components can access to their instance data using the `DATA` construct. The structure of this construct is given below.

```
ACCESSDATA ::= "DATA." fieldName
```

Access to attributes Components can access to their attributesusing the `ATTRIBUTES` construct. The structure of this construct is given below.

```
ACCESSATTR ::= "ATTRIBUTES." fieldName
```

5.3 C++

This feature is not supported yet in the development mainstream. The documentation will be added once the C++ will be supported by a mainstream development language. Note that a component implementation may not have a global variable.

6 Examples

6.1 YUV Player

This section aims at illustrating some basic features for programming with THINK . To this end , we present how basic multimedia player application that displays on screen a YUV video can be built and configured in THINK .

6.1.1 Getting started

The example presented in this section can be found in the folder `kortex/examples/yuv-player`. It can be built with classical `ant` commands and its executable result or kernel image is produced in `build/<target>/<target-component>` file. The executable needs a film in CIF 4:2:0 format. We suggest to download a film conforming to this format at *http://eeweb.poly.edu/ yao/VideobookSampleData/video/MPEG4/singer/singer_cif_ori90.zip*. The current version of the player is programmed to find the film in the base path where it is launched.

6.1.2 Architectural overview

As depicted in figure 4, our player is composed by three high-level components: (1) a main player that receives a RGB stream to be displayed on screen, (2) a filter pipeline that implements some stream transformation operations at least for transforming the input YUV stream to an RGB stream and finally (3) a file reader component that reads the stream to be used by the rest of the application from a file. Note that the filter pipeline is modeled as a composite component which is composed by basic filter components such as YUV to RGB transformer and so on. Note also that the player application itself is modeled by a composite component including above components.

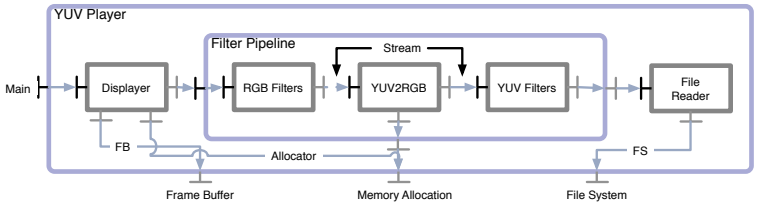

Figure 4: Component-based architecture of the YUV player.

The interconnection of the subcomponents of the YUV player is insured by a single interface type, called `streaming.api.Stream` (Figure 6). A typical *stream processor component* is indeed a filter providing an `outstream` interface and requiring an `instream` interface of type

```
<definition name="StreamProcessorType">
  <interface name="outstream" signature="streaming.api.Stream" role="server" />
  <interface name="instream" signature="streaming.api.Stream" role="client" />
</definition>

<definition name="Source">
  <interface name="outstream" signature="streaming.api.Stream" role="server" />
</definition>

<definition name="Consumer">
  <interface name="instream" signature="streaming.api.Stream" role="client" />
</definition>
```

Figure 5: Architectural defiinition of stream processor, source and consumer component types.

`streaming.api.Stream` and implementing a transformation operation between them. A typical *source* component such as the *reader* implements only an `outstream` interface. A typical *consumer* component such as the *displayer* only requires an `instream` interface. The communication model that has been adopted in this application is can be qualified as *pull* in the sense that each component use the `getFrame` method of their `instream` interface to pull a frame of streaming data.

```
package streaming.api ;

public interface Stream {
  int getFrame(byte[] frame) ;
  int getWidth() ;
  int getHeight() ;
  int getFrameRate() ;
  int getFrameBufferSize() ;
  boolean hasFrames () ;
}
```

Figure 6: Definition of `streaming.api.Stream` interface.

As the most of the application software, the entry point of this software is an interface of the `activity.api.Main`. Beside this server interface, the player application needs some system services such as framebuffer display, memory allocation and file system to implement the video display from an input file. These dependencies are expressed by the client interfaces of the application which are respectively called `fb`, `allocator` and `fs`.

Figure 7 presents the ADL definition of the above YUV player application. This architecture definition starts defining the server and client interfaces of the application. The lines 6 to 20 define the subcomponents of the application. Note that the filterPipe composite's content is directly specified in this description. That is the reason why the architecture definition of this component contains itself interface, subcomponent and binding definitions. Finally, the lines 20 to 28 defines the functional and system dependency bindings between the subcomponents of the application.

6.1.3 Implementation details

Let's take a look at the implementation of the displayer component implementing the main interface to have a better understanding of the execution of the player application. As depicted in Figure 8,

```
<definition name="YUVPlayer">
  <interface name="main" signature="activity.api.Main" role="server" />
  <interface name="fb" signature="video.api.FrameBuffer" role="client" />
  <interface name="allocator" signature="memory.api.Allocator"
    role="client" />
  <!-- Main player component -->
  <component name="main" definition="Player" />
  <!-- Filter pipeline implementing stream transformation operations-->
  <component name="filterPipe" definition="StreamProcessorType">
    <interface name="allocator" signature="memory.api.Allocator"
      role="client" />
    <!-- Contains only YUV t RGB transformer -->
    <component name="yuv2rgb" definition="YUV2RGB" />
    <binding client="this.outstream" server="yuv2rgb.outstream" />
    <binding client="yuv2rgb.instream" server="this.instream" />
    <binding client="yuv2rgb.allocator" server="this.allocator" />
  </component>
  <!-- File reader component -->
  <component name="yuvReader"
    definition="reader.FileReader(352,288,15,singer_cif_ori90.yuv)" />

  <!-- Definition of the application pipeline -->
  <binding client="main.stream" server="filterPipe.outstream" />
  <binding client="filterPipe.instream" server="yuvReader.outstream" />
  <!-- Export/Import connections-->
  <binding client="this.main" server="main.main" />
  <binding client="main.fb" server="this.fb" />
  <binding client="main.allocator" server="this.allocator" />
  <binding client="filterPipe.allocator" server="this.allocator" />
</definition>
```

Figure 7: Architectural definition of the YUV player application.

the implementation of the displayer component consists in the implementation of the single `main` method of its server interface called also `main`. First, the displayer component does not have any state variables since the life cycle of this component is the same as the life cycle of the `main` method. The implementation of this method starts by initializing some features related to the frame buffer and allocating memory for storing the input frame. Once this initialization step is achieved, it loops until the end of the frames for copying each frame into the virtual screen of the frame buffer for displaying the images. At the end of each copy operation, the frame buffer is flushed to insure that the content of the virtual screen be displayed on screen.

As mentioned above, the stream processor components are filters providing and receiving frames over stream interfaces. Figure 11 illustrates the architecture definitions for two different stream processor components. The first one is he filter that transforms a YUV stream to a RGB stream. In addition to the functional `instream` and `outstream` interfaces, this component needs a memory allocation service and provides a life cycle controller interface to initialize its state before becoming functional. Note that is a component provides a life cycle components, one can be sure that the `start` method of this interface will be invoked at the construction of the component by the deployment infrastructure. Similarly, the `stop` interface will be invoked before the component be destroyed. In this sense, this interface can be considered as being the constructor and the destructor of a component. Th allow the deployment environment for finding the life cycle interface of a component, this interface must conventionally by called "lifecycle-controller". Note that the "-" characters are transformed into "_" in implementation to be conform to C/C++ syntax.

```
DECLARE_DATA {};

#include <think.h>

void METHOD(main, main)(void* _this, jint argc, char** argv) {
  char *fb_base;
  int height, width;
  int i, j;
  int depth;
  jbyte *frame;
  int screen_h, screen_w;
  int frame_size;
  int x, y;

  // Initialize the screen
  CALL(REQUIRED.fb, clearscreen);
  // Get the framebuffer base address
  fb_base = CALL(REQUIRED.fb,address);
  // Prepare display properties
  width = CALL(REQUIRED.stream, getWidth);
  height = CALL(REQUIRED.stream, getHeight);
  frame_size = CALL(REQUIRED.stream, getFrameBufferSize);
  screen_h= CALL(REQUIRED.fb, height);
  screen_w = CALL(REQUIRED.fb, width);
  // Allocate memory for the input frame
  frame = (jbyte *)CALL(REQUIRED.allocator, alloc, frame_size);

  // Pull frames to display until the end of stream
  while(CALL(REQUIRED.stream, hasFrames)) {
    CALL(REQUIRED.stream, getFrame, frame);
    j = 0;
    // Display the frame on the framebuffer
    for(y=0; y<height; y++) {
      for(x=0; x<width*4; x+=4) {
        fb_base[x+(y*screen_w*4)+0] = 0;
        fb_base[x+(y*screen_w*4)+1] = frame[j++];
        fb_base[x+(y*screen_w*4)+2] = frame[j++];
        fb_base[x+(y*screen_w*4)+3] = frame[j++];
      }
    }
    CALL(REQUIRED.fb, flush, 0, 0, width, height);
  }
}
```

Figure 8: Implementation of the top level YUV player.

In the same way, the second definition describes another stream processor component transforming a color YUV stream to a grayscale YUV stream. The architecture of this component is simpler than the first one since it has no additional system dependencies.

Now take a look at how this grayscale transformer processor is implemented. As usual (Figure 10), the first part of the implementation file is used for declaring the state data of the component. The state data of the grayscale component includes some informations on the properties of the frames that are transmitted in the stream. The values of these variables are initialized in the `start` method of the life cycle controller interface (at the bottom of the figure). The rest of the method implementations involve the implementation of the `outstream` server interface. Note that, most of these methods have trivial functionalities since they returns values that are depicted from the input stream, without transformation. The only single method that proceeds to a transformation is the `getFrame` method

```
<definition name="YUV2RGB" extends="StreamProcessorType">
  <interface name="allocator" signature="memory.api.Allocator" role="client" />
  <interface name="lifecycle-controller" signature="fractal.api.LifeCycleController"
             role="server" />
  <content class="yuv2rgb" language="thinkMC" />
</definition>

<definition name="Grayscale" extends="StreamProcessorType">
  <interface name="lifecycle-controller" signature="fractal.api.LifeCycleController"
             role="server" />
  <content class="grayscale" language="thinkMC" />
</definition>
```

Figure 9: Definitions of two filter components which respectively transforms a YUV stream to a RGB stream, and transforms a colored YUV stream to a grayscale YUV stream.

which is in charge of setting to *zero* a given part of the stream to eliminate the color information. By this way, the input stream containing colors is transformed to a grayscale frame.

6.1.4 Static architecture reconfiguration exercise

Once the functional principals of the player application has been understood, one can try to modify the filter pipeline to change the behavior of this application. To this end, we provide a set of filters for grayscale transformations, resizing and blur effects. Readers can also write their own filters to out hands in the implementation issues.

Figure **??** illustrates the modified version of the architecture definition of the player application depicted in figure 7 to insert the grayscale transformer in the filter pipeline where the YUV stream is transmitted. Note that the place where a filter may be inserted depends on the type of Stream (in our case YUV or RGB) that it consumes and produces.

6.1.5 Support for different target platforms

7 Other documentation resources

```
DECLARE_DATA{
  int width ;
  int height ;
  int frame_size ;
  int u_offset ;
  int v_offset ;
};


#include <think.h>

jint METHOD(outstream, getFrame)(void *_this, jbyte* frame) {
  int y_size = DATA.height*DATA.width ;
  CALL(REQUIRED.instream, getFrame, frame) ;
  /* Put null value for u and v to eleimnate the colors */
  memset(frame+y_size, 128, DATA.frame_size - y_size) ;
}

jint METHOD(outstream,getWidth) (void *_this) {
  return DATA.width ;
}

jint METHOD(outstream,getHeight) (void *_this) {
  return DATA.height ;
}

jint METHOD(outstream,getFrameRate) (void *_this) {
  return CALL(REQUIRED.instream, getFrameRate) ;
}

jint METHOD(outstream,getFrameBufferSize) (void *_this) {
  return DATA.frame_size ;
}

jboolean METHOD(outstream,hasFrames) (void *_this) {
  return CALL0(REQUIRED.instream, hasFrames) ;
}

void METHOD(lifecycle_controller, start) (void *_this) {
  GETSELF ;
  DATA.width = CALL(REQUIRED.instream, getWidth);
  DATA.height = CALL(REQUIRED.instream, getHeight);
  DATA.frame_size = CALL(REQUIRED.instream, getFrameBufferSize) ;
  DATA.u_offset = DATA.width * DATA.height ;
  DATA.v_offset = 5*DATA.width * DATA.height/4 ;
}

void METHOD(lifecycle_controller,stop) (void *_this) {} ;
```

Figure 10: Implementation of the top the grayscale filter player.

```
<component name="filterPipe" definition="StreamProcessorType">
  <interface name="allocator" signature="memory.api.Allocator"
    role="client" />
  <!-- Contains only YUV t RGB transformer -->
  <component name="yuv2rgb" definition="YUV2RGB" />
  <component name="grayscale" definition="Grayscale" />
  <binding client="this.outstream" server="yuv2rgb.outstream" />
  <binding client="yuv2rgb.instream" server="grayscale.outstream" />
  <binding client="grayscale.instream" server="this.instream" />
  <binding client="yuv2rgb.allocator" server="this.allocator" />
</component>
```

Figure 11: Definitions of two filter components which respectively transforms a YUV stream to a RGB stream, and transforms a colored YUV stream to a grayscale YUV stream.