Learning THINK from a simple example: user & developer view

Juan NAVAS
April 2008
summary

- PART I : THINK Fundamentals
  - FRACTAL component based model
  - THINK as a FRACTAL model implementation

- PART II : THINK in action
  - using THINK
  - extending the framework
PART I

THINK FUNDAMENTALS
THINK [1] in short

- FRACTAL [2] component-based model implementation
- Components functional code written in "C"
- Languages
  - ADL : Architecture Description Language (human-readable style)
  - IDL : Interface Description Language
  - nuptC = C language + annotations
- Architecture compiler
- Components library (i.e. KORTEX)
- Open source project (LGPL)
FRACTAL component model

(For details, please see [2])

- Development & Run–Time component model
- Components models advantages in embedded systems design & development:
  - Facilitates programming of systems' dynamic behavior and adaptability needs
  - Code reuse: components are independent entities, and are the units of administration and deployment
  - Parallel, fast development
  - Separation of concerns (functional vs. non–functional)
FRACTAL component model (2)

- Reflexive model
  - Introspection: discover system’s constitution
  - Intercession: modify system’s architecture
- Recursion
  - A component can be made of other components
- Implementations
  - Java: Julia [3]
  - C: THINK
  - Others (i.e. .NET, SmallTalk)

(For details, please see [2])
FRACTAL/THINK component model

- A component:
  - Provides server interfaces
    - Services offered to other components
  - Requires external interfaces (client interfaces)
    - Required services that shall be offered by a different component
  - Has attributes
    - Components internal variables that shall be exposed
A component has also a content and a membrane

- **The content:**
  - Implements server interfaces ("C" code)
  - Can enclose other components (subcomponents)

- **The membrane:**
  - Implements control interfaces (i.e., BindingController)
  - Has its own content
  - Is facultative
• **Bindings:**
  
  • Are communication channels from client to server interfaces
  
  • Also model “service transferring” between a component server interface and a subcomponent server interface of the same type
THINK – building FRACTAL systems

- High Level Compiler
  - Allows high-level architecture transformations and optimizations (i.e. static bindings, unique components)
  - Transforms & generates source code based on non-functional properties

- Low Level Compiler
  - Uses gcc to produce a system’s binary image from an architectural description (ADL’s files) and functional source code (C files)

- An extensible Compiler
  - Compiler architecture is also component-based
  - Follows the FractalADL model [4]
  - Written in Java
PART II
THINK IN ACTION
an application example

- Communication from an AT91SAM7X–EK board (ARM7) to a PC, via debug port
user view – designing the architecture

- Component *boot* represents the system entry point
- Component *console* is an abstraction of all visual representation mechanisms
- Component *printer* controls serial debugger port (hardware resource)
user view – designing the architecture – ADL

```
component example {

contains boot = boot.boot
contains console = console.console
contains printer = printer.printer

binds boot.cons to console.cons
binds console.print to printer.print
binds console.checkPrinter to printer.check
}
```

console is a subcomponent of example

console's client interface is bound to printer server interface
user view – designing the architecture – ADL (2)

```
component printer.printer {
  provides api.print as print
  provides api.check as check

  contains  checker = printer.printerchecker
  binds  this.check  to  checker.check

  content  printer.printer
  content  hardware.dbgu.dbgu raw
  content  hardware.io.lib_AT91SAM7X256 raw
}
```

Server interface `print` definition is in file `api/print.idl`

`printer/printer.c` contains server interfaces implementations

This file is included as it (directly linked) in binary image
user view – interfaces

- Components’ interfaces are specified in a pseudo IDL language, a Java/C mix
- *print*-type interface declaration (*api/print.idl* file):

```java
package api;

public interface print {
    int printk( char* message );
}
```

This type of interface only includes a method definition, but it could have many of them.
user view – components content

- **nuptC** language is used to program functional code
- nuptC is C language + annotations [6]
- Legacy code can be easily imported adding annotations, they are considered as comments (and thus ignored) by common C compilers

- *printer.c* (extract):

```c
/* dbg from hardware */
extern void AT91F_DBGU_Printf(char *buffer);

// @@ DefaultServerMethods(print) @@ //
int printk(char* pMessage) {
    AT91F_DBGU_Printf(pMessage);
    return 1;
}
```

- 10+ annotations (see technical documents for further information)
an architecture modification example: protection of critical components

- **printer** component manages the debug serial port, which is a critical hardware resource
- **printer** content has access to critical data, hence access to its server interfaces must be supervised by another agent
- **printer** component shall be placed in **kernel** space
- **printer** server interfaces shall be accessed only via **system calls**
user view – how?

- Easy, using THINK’s properties [6]

```plaintext
component printer.printer [kernel = true] {
  provides api.print as print
  provides api.check as check
  contains checker = printer.printerchecker
  binds this.check to checker.check
  content printer.printer
  content hardware.dbgu.dbgu raw
  content hardware.io.lib_AT91SAM7X256 raw
}
```

THINK compiler will understand that printer component’s content must be placed in kernel space and that its server interfaces shall be accessed via system calls, and generate the proper stubs.
user view – results

- *example* ADL is passed through THINK’s “machinery” (compiler) and a proper exe file is generated
- New C code is also generated, corresponding to new components added by the compiler
developer view – how?

- A little less easy ;-)  
- THINK’s high-level compiler is “FRACTALized” and contains a chain of **Loaders**, each one managing a certain aspect of system’s description.
developer view – how ? (2)

- A new loader that reacts to **kernel** property existence shall be created: **SysCallsLoader**

- As THINK is written in Java, a simple class is to be created: **SysCallsLoader.java**
developer view – the new loader

- **Loader** server interface, included in all Loader-style components, includes method `load`, where all the work is done:

```java
public Definition load(String name, Map context) throws ADLException {
    Definition d = clientLoader.load(name, context);
    myLoad((ComponentContainer) d, context);
    return d;
}
```

A `Definition` object represents a system's AST.

The AST is retrieved from the lower-level loader in loaders chain.
**SysCallsLoader** requires a **nodeFactory** client interface to create new components and interfaces

```java
Component comp = (Component) nodeFactoryItf.newXMLNode(
    "classpath://org/objectweb/think/adl/parser/xml/think.dtd",
    "component");

TypeInterface cltItf = (TypeInterface) nodeFactoryItf.newXMLNode(
    "classpath://org/objectweb/think/adl/parser/xml/think.dtd",
    "interface");
```
developer view – implementing new interfaces

- New components ⇒ new interfaces ⇒ methods implementations ⇒ new source code
- Implementations are created as any other FRACTAL object, another Java class object is specified to generate them

```
ExtImplementation impl = (ExtImplementation) nodeFactoryItf.newXMLNode(  
    "classpath://org/objectweb/think/adl/parser/xml/think.dtd",  
    "implementation");
impl.setName("SysCallInterceptor");
impl.setLanguage("nuptse");
impl.setClassName("org.objectweb.think.primitive.nuptse.controllers.syscalls.SCInterceptorInfo");
```

SCInterceptorInfo.java generates source code for this particular component
developer view – CodeGen

- In order to generate source code, this new Java class uses CodeGen [6]
- CodeGen is a Java package for:
  - Code generation from scratch
  - Code transformation (parsing, handling and production)
- CodeGen is also LGPL and it is included in THINK releases
developer view – CodeGen (2)

```java
Function scFunction = ((Function) sc.getFunction()).makeCopy("SRV_" +
    itf.getName() + "_" + sc.getName());
FunctionBlock scBlock = scFunction.getFuncBlock();
CType returnType = ((FunctionT) scFunction.getType()).getReturnType();
CVariable r = returnType.newVar("r");
scBlock.add(r.getDeclaration());
scBlock.add(new Assignment(r, new IntC(1)));
scBlock.add(new Return(r.getExpression()));

int SRV_print__printk (char *message) {
    int r;
    r = 1;
    return r;
}
```

* pseudo-code
SysCallsLoader is integrated to the loaders chain, architecture is transformed, and our example can be compiled.

```
int SRV_switrap (int n) {
    int r;
    asm volatile ("mov r1, %0" : : "r" (n));
    asm volatile ("swi 0x0" : :);
    asm volatile ("mov %0, r0" : =r (r) : );
    return r;
}

int SRV_print_at_printer_printk(char *message) {
    int r;
    // Saving method inputs
    ((sArgsArray)[0] = ((int) message));
    (r = (*cltIfsId->swi->methDesc->trap)(0));
    return r;
}
```
by this example we have seen...

- THINK's ADL programming: bindings, composition, interfaces...
- NuptC programming: annotations
- THINK's properties: non-functional aspects
- How to develop a new architecture modifier
  - Loaders
  - Codegen
THINK is an easy way to adopt Component Based System Engineering (CBSE)
- Transparent use of legacy code via annotations
- Intuitive ADL & IDL
- Existent components libraries (i.e. KORTEX) in various platforms (ARM, MSP, AVR, PowerPC...)

THINK is an extensible framework, and it’s relatively simple to extend it
- FRACTAL based
- Written in Java language
- CodeGen package included

THINK is free software
- [http://think.objectweb.org/download.html](http://think.objectweb.org/download.html)
balance

- Plenty of not covered topics!
  - Dynamic behavior of FRACTAL/THINK run-time models
    - Next: "Hands-on THINK's Control Interfaces"
  - Global extensions: Aspect Oriented Programming (AOP) approach applied to architectural descriptions
  - System Calls example low-level code
references


