Component Based Software Engineering with Fractal/Cecilia

Alessio Pace

INRIA

Agenda

- Component Based Software Engineering (CBSE)
- The Fractal Component Model
- Cecilia: a C implementation of Fractal
  - Overview
  - HelloWorld example
  - Tools & libraries

Component Based Software Engineering (CBSE)

Software Complexity

- A software may have challenging requirements in terms of:
  - development
  - deployment
  - integration
  - (re-)configuration
  - maintenance / evolution
  - modularity / flexibility
  - heterogeinity of some of the parts
  - legacy elements
  - quality of service
  - ...

Component Based Software Engineering

- An effort towards the rational construction and management of complex software system
- Dealing with the modular construction of software systems by means of the explicit composition of software units (**components**), and importance given to **system architecture**

Expected benefits of CBSE

- Adaptation
- Integration
- Interoperation
- Management (administration)
- Prediction of chosen properties
  - reliability, security, scalability, testability, QoS, ...

Common concepts among various components models

- Components
  - runtime entities with contractually specified interfaces
  - written in programming languages such C, C++, Java, ...
- Interfaces (aka ports)
  - client (required) / server (provided) interfaces
  - usually defined using an Interface Definition Language (IDL)
- Bindings (aka connectors)
  - connectors among interfaces of the components
  - various types: synchronous/asynchronous, local/remote, ...
- Architecture Description Language (ADL)
  - to define relationships among components and their properties

Concepts depending on the component model and implementation language #1/2

- ADL
  - dedicated language (ie. XML based) or using constructs of the programming language (ie. Java 5 Annotations)
- Hierachical (*composite*) components
  - only at design time (in the ADL)
  - also at runtime time (**primitive** and **composite** components)
- Introspection and intercessions capabilities
  - possibility to discover components interfaces and relationships among them
  - possibility to reconfigure the architecture at runtime
  - fixed (by the model) / open reflection capabilities
  - in general, how much they can be controlled, introspected, instatiated, destroyed

Concepts depending on the component model and implementation language #2/2

- Multi-language support
  - implementations for different programming languages
  - interoperability among components written in different languages
    - ie. through an ORB
- Programming language invasiveness
  - mandatory interfaces to implement / classes to extend
- Interface Definition Language (IDL)
  - dedicate language (ie. Corba IDL) or using constructs of the language used to write components code (ie. Java Interfaces)
- Container
  - do components need to be deployed on a container?

The Fractal Component Model

Reasons for the Fractal Component Model

- Limitations in other component models and ADLs:
  - limited support for extension and adaptation
  - fixed forms of composition
  - fixed forms of introspection & intercession
- « Develop a powerful (**reflective**) but **flexible** / **extensible** / **customisable language independent** component model for any kind of software (from middlewares to operative systems) throughout the complete software lifecycle, with an enphasis on **runtime reconfiguration** and **management** which is in general the least well handled parts in existing component models. »

The Fractal Component Model #1/2

- **Open**
  - extra-functional services associated to a component can be customized through the notion of a **control membrane**
- **Recursive**
  - components can be nested in **composite components**
  - uniform view at any level of the system architecture
- **Execution Model Independant**
  - **no execution model is imposed**. Components can be run within other execution models than the classical thread-based model such as event-based models and so on
- **Language agnostic**
  - existing implementations for **various programming languages** (Java, C, ...)

The Fractal Component Model #2/2

- **Component Sharing**
  - a given **component instance can be included (or shared) by more than one component**. This is useful to model shared resources such as memory manager or device drivers for instance
- **Binding Components**
  - a single abstraction for components connections that is called bindings. **Bindings can embed any communication semantics** from synchronous method calls to remote procedure calls
- **Selective reflection**
  - components *can* have **full introspection and intercession capabilities**
  - different components in the same architecture may have **different level of introspection and intercession**

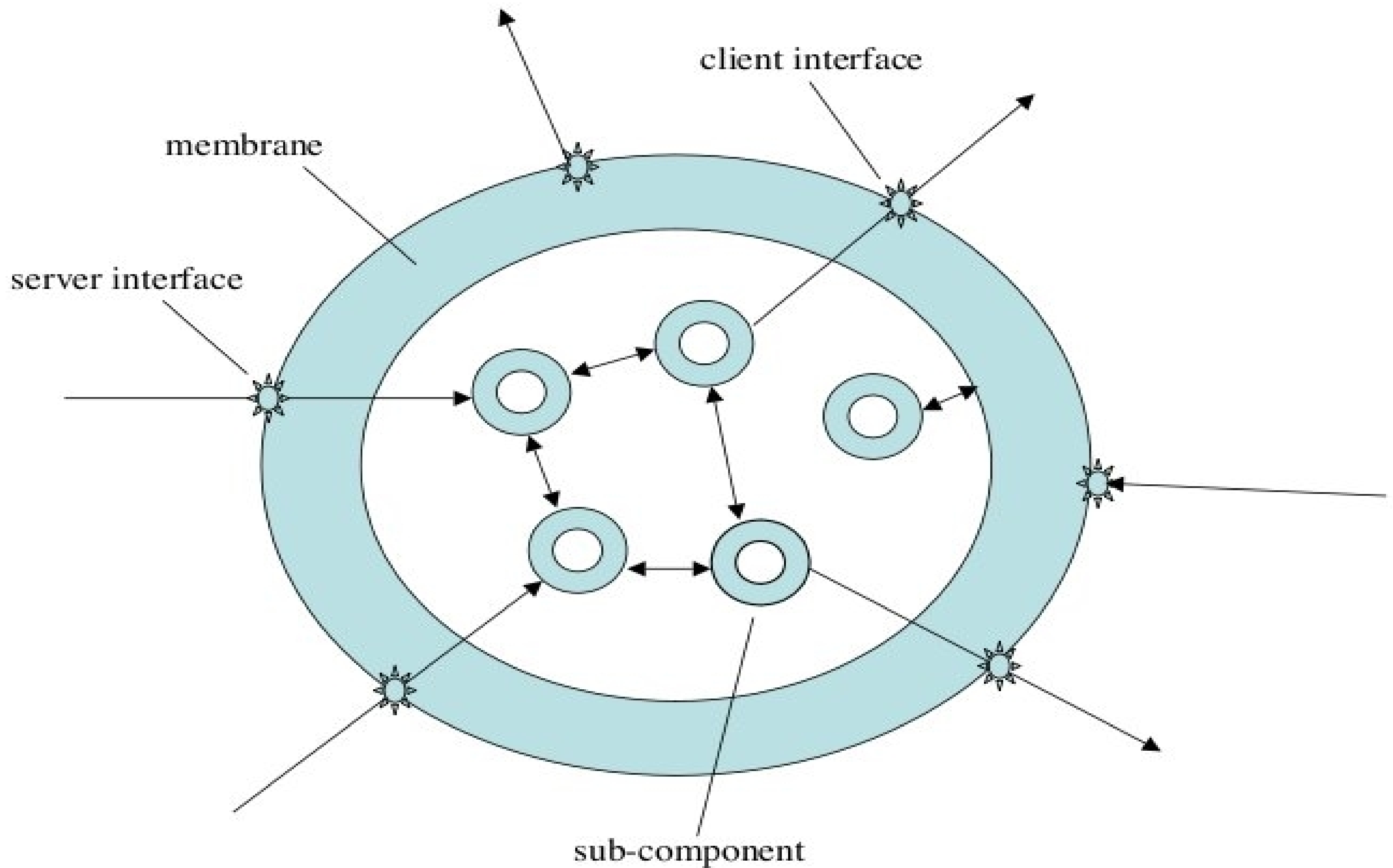
Interpretation of "classical" concepts

- **Components**
  - runtime entities, not only design time / deploy time
  - made of **membrane + content**
- **Interfaces**
  - the only access points to components
  - client (required) / server (provided) interfaces
  - emit and receive operation invocations
- **Bindings**
  - no fixed semantics
  - **primitive** bindings: in the same address space (ie. an object reference)
  - **composite** bindings: for distributed (ie. RMI) or heterogeneus (ie. JNI) communication

The membrane

- **Composition** and **reflection** behaviour
- Can provide access to reflection capabilities via **controller interfaces**
- **No fixed set of controllers** for component introspection and intercession
- *Can* have an internal structure of its own
- **No fixed semantics**
- Can have interceptors
- Components in the same architecture can have different membrane structure

Components: membrane + content

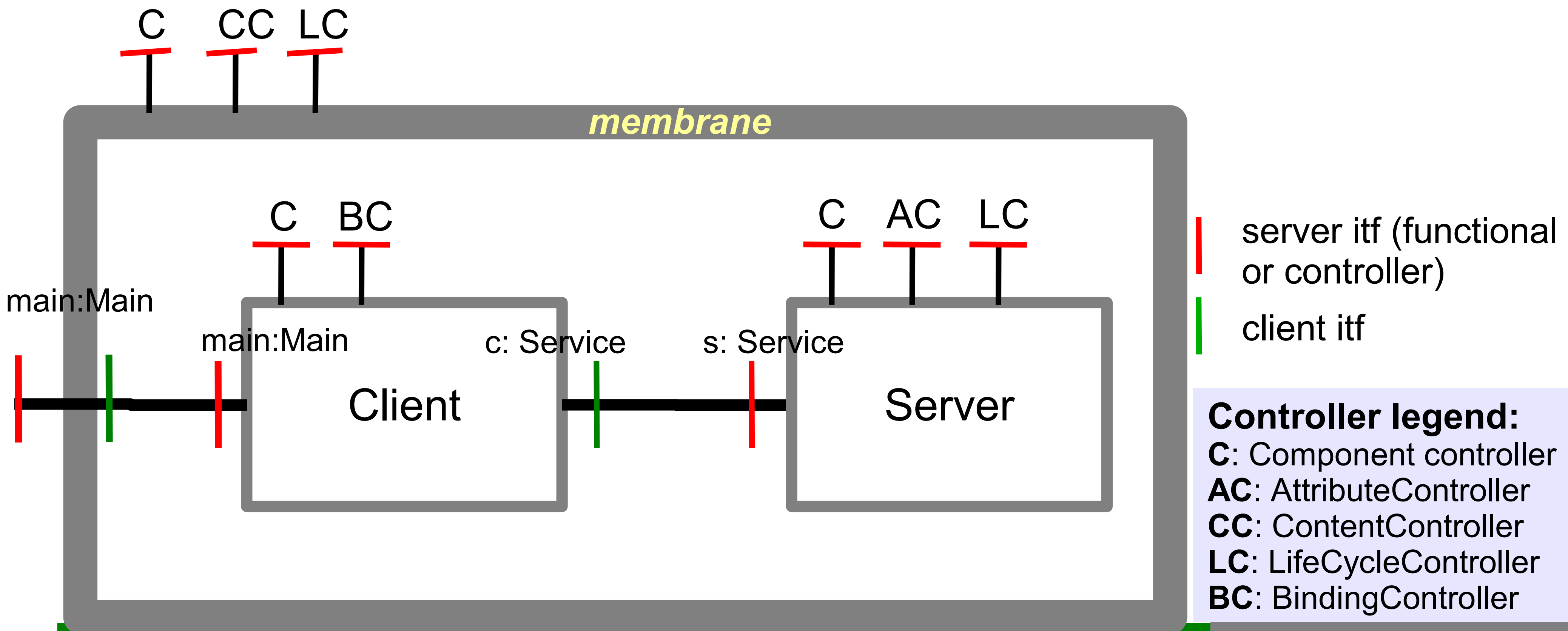


Standard Controllers

- Reflection : minimal
  - **Component controller** (discovering component interfaces)
  - **Binding controller** (binding an external component interface)
- Reflection : structural
  - **Content controller** (adding, removing subcomponents)
  - **Attribute controller** (setting, getting component attributes)
- Reflection : behavioral
  - **Lifecycle controller** (starting, stopping the component)

A Fractal example: HelloWorld

- HelloWorld composite component with
  - *external* interfaces: exposed to the exterior
  - *internal* interfaces: to 'export' sub components' interfaces
- Client and Server primitive components



Fractal: conclusions

- From objects to reflective components to build manageable systems
  - Interfaces
  - Explicit connections
  - Membranes (reflective components)
- Computational model for open systems
  - open binding semantics
  - open reflection semantics
- Extensible ADL & associated toolchain
- More info on the website
  - http://fractal.ow2.org/

Cecilia

Cecilia #1/2

- **Cecilia** is an *environment* for programming Fractal components in the C language
- It is based on work made on Think and Plasma projects
- Components implementation is written in **C** (with commodity macros)
- Components interfaces are expressed in a Java-like Interface Definition Language called **Cecilia IDL**
- Components architecture is described in the Fractal ADL XML-based language, and built by the associated **Fractal ADL toolchain**

Cecilia #2/2

- **Open Source license** (LGPL)
- Available on the OW2 SVN repository of the Fractal project
  - svn://svn.forge.objectweb.org/svnroot/fractal/trunk/cecilia-framework
- **Actively developed**
  - STMicroelectronics
  - INRIA Sardes
- **Documentation**
  - http://fractal.ow2.org/cecilia-site/current/
- Support on the **Fractal mailing list**
  - fractal@objectweb.org

The Cecilia programming environment #1/2

- From the Cecilia home page:
  - *The Cecilia toolchain reads a set of Fractal ADL architecture description files defining the configuration of the components a given software system, and builds the corresponding software by gathering the components to be used 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*
  - the generated code conforms to **standard C**, and the tools used to compile both the implementation and generated files are standard C or C++ compiler and linker for a given target hardware platform

The Cecilia programming environment #2/2

- What the developer has to write
  - ADL files describing the application components architecture
  - IDL files to define the interfaces in use by the components
  - C code (using commodity macros) for the functional interfaces of *primitive* components, allowing to concentrate only on writing component-related aspects such as instance data, interface implementations, invocations and so on
- What is automatically generated and done by the toolchain
  - generation of C code for the controller interfaces of primitive components
  - generation of C code for composite components
  - binding of the component interfaces
  - source code compilation and linkage by reusing standard C compilers and linkers for the target platform

The Cecilia toolchain workflow

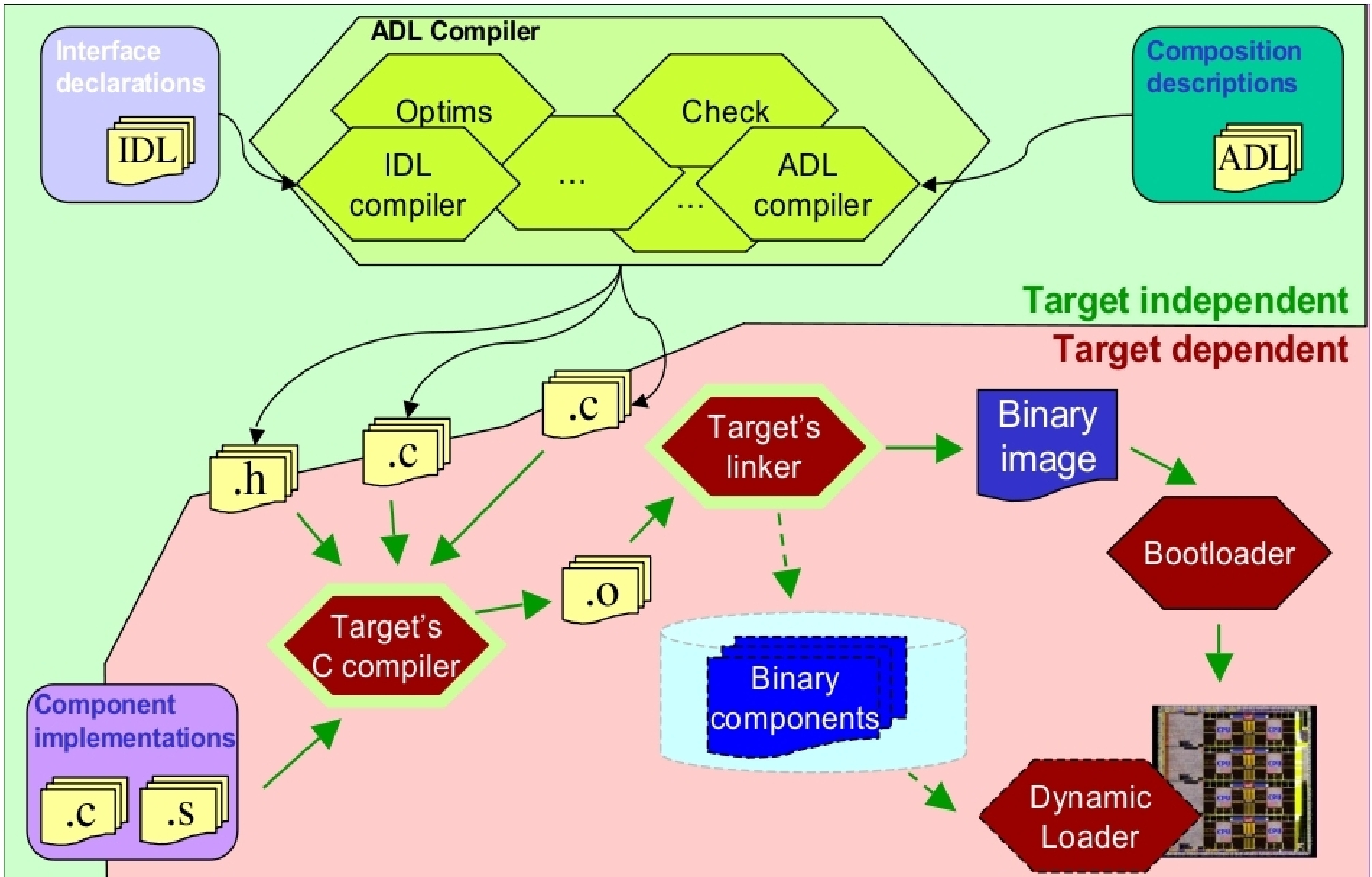


image property of ST Microelectronics

The Cecilia environment

- **Toolchain written in Java** and capable of building software systems from their architecture descriptions (**Fractal ADL** files) and component interfaces (**Cecilia IDL** files)
- **Fractal APIs** written in the **Cecilia IDL** language
- The **C** implementation of the **Fractal component model** written in the C language
- Compilation / dependency resolution tools
  - either via **Maven + Cecilia maven tools**
  - or via **Makefile + ceciliac** (shell wrapper for the Cecilia compiler)

Cecilia HelloWorld

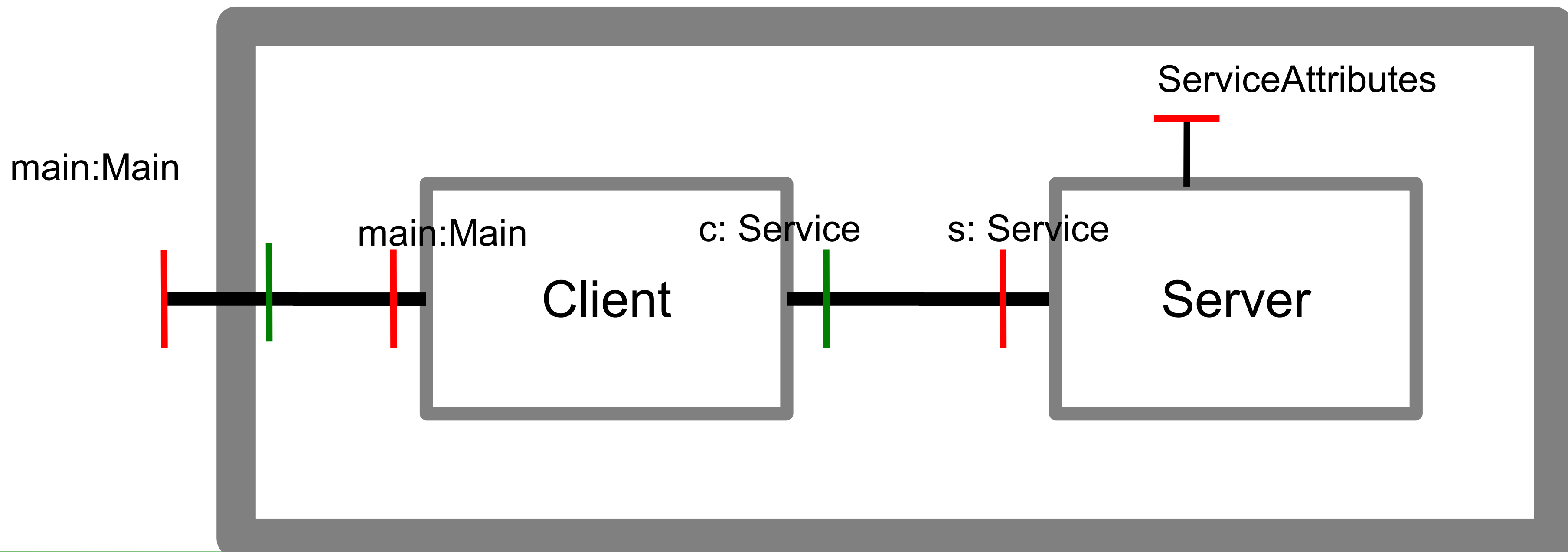
hands first tutorial

Cecilia HelloWorld tutorial

- Top-down overview of
  - Fractal ADL language to define components architecture
  - Cecilia IDL language to define components interfaces
  - Cecilia language to program primitive components
- Available in its original HTML format at
  - http://fractal.ow2.org/cecilia-examples-site/current/helloworld/index.html
  - http://fractal.ow2.org/cecilia-examples-site/current/helloworld/running.html

HelloWorld architecture

- Composite component with a Main interface
  - every Cecilia application must provide as entry point a 'main' interface of type boot.api.Main
- Client component providing Main and requiring Service
- Server component providing Service and parametrizable with the ServiceAttributes



Step #1

Architecture definition in the Fractal ADL language

HelloWorld architecture

```
<definition name='HelloWorld'>
  <!-- mandatory 'main' name for boot.api.Main entry point -->
  <interface name='main' role='server'  signature='boot.api.Main' />

  <component name='client'>
    <interface name='m' role='server' signature='boot.api.Main' />
    <interface name='c' role='client' signature='hw.Service' />
    <content class='hw.client' />
  </component>

  <component name='server'>
    <interface name='s' role='server' signature='hw.Service' />
    <content class='hw.server' />
    <attributes signature='hw.ServiceAttributes'>
      <attribute name='header' value='prefix:' />
    </attribute>
  </component>

  <binding client='this.main' server='client.m' />
  <binding client='client.c' server='server.s' />

</definition>
```

Fractal ADL

- The Fractal XML-based extensible Architecture Definition Language
  - different modules to cover different aspects: components definition, their interfaces, bindings among them, attributes (properties), component containment, component content, component remote deployment, component definition extension from another definition, ...
  - new modules can be added to cover other aspects
    - ie. BindingFactory module to allow bindings (client/server interfaces) over arbitrary communication protocols
- The language grammar is defined by means of an XML DTD (Document Type Definition)

About the ADL language used in Cecilia

- A small *extension* of the base Fractal ADL
  - http://fractal.ow2.org/cecilia-site/current/toolchain-parent/cecilia-adl/language.html
- To cover additional aspects specific to C
  - cflags
  - ldflags
- **Open** for further extensions

```
<definition name='HelloWorld'>
  <!-- -->
  <component name='server'>
    <interface name='s' role='server' signature='hw.Service' />
    <content class='hw.server' >
       <cflag value='...' />
       <ldflag value='...' />
    </content>
  </component>
  <!-- ... -->
</definition>
```

Composition and binding rules

- Top level component in an ADL file as a <**definition>** element
- Sub components as **<component>** sub elements
- Top level or sub components declare their **<interface>**s
  - name, role ("client" | "server"), signature
- Primitive components declare their implementation artifact
  - **<content class="path.to.implFile" />**
  - type of the artifact depends on the Fractal implemetation (Java class, C file, ...)
- A composite component declare **<binding>**s for its direct sub components and its internal interfaces

Step #2

Interfaces definition in the Cecilia IDL language

HelloWorld interfaces

```
/* This interface is defined in the cecilia-baselib library */
package boot.api;

interface Main {
  void main(int argc, char* argv[]);
}
```

```
/* This interface is defined in the HelloWorld example */
package hw;

interface Service {
  void print(string message);
}
```

```
/* This record is define in the HelloWorld example*/
package hw;

record ServiceAttributes {
  string header;
}
```

About the Cecilia IDL language

- **Strongly typed**. The compilation toolchain can verify the compatibility of interconnected component interfaces
- Supports two types of constructs
  - **interfaces**:
    - to define the methods provided by a component interface
  - **Records** (for two purposes):
    - to define a Fractal AttributeController, in order to `{get|set}` properties on a component
    - to define a C struct which can be passed to or returned from interface methods
- Documentation online
  - http://fractal.ow2.org/cecilia-site/current/toolchain-parent/idl-parser/language.html
  - http://fractal.ow2.org/cecilia-site/current/toolchain-parent/idl-parser/idl-to-c.html

Step #3

Primitive components implementation

The Client primitive component

```
DECLARE_DATA {
  // component instance variables may go here
};

/* Always include cecilia.h after the DECLARE_DATA section */
#include <cecilia.h>

/* Implementation of the 'main' method of the 'm' server itf of
 * type 'boot.api.Main' (IDL interface seen before).
 */
int METHOD(main, main)(void* _this, int argv, char* argc[]) {

  /* Get a reference to the Service client interface */
  hw_Service serviceItf = REQUIRED.c;

  /* Call the print method on the serviceItf reference
   * passing a parameter */
  CALL(serviceItf, print, ''from client'');

  return 0;   // ok
}
```

The Server primitive component

```
DECLARE_DATA {
  // component instance variables may go here
};

/* Always include cecilia.h after the DECLARE_DATA section */
#include <cecilia.h>

/* Implementation of the 'print' method of the 's' server itf
 * of type 'hw.Service' (IDL interface seen before).
 */
void METHOD(s, print)(void *_this, char *msg) {

  /* access the value of the header attribute as it has been
   * configured in the ADL */
  char* header = ATTRIBUTES.header;
  if (header == NULL) {
    header = "";
  }

  printf("%s%s\n", header, msg);
}
```

About the Cecilia language for programming primitive components

- A set of C macros to code primitive components
  - **DECLARE_DATA** : to define the component instance datas
  - **METHOD** : to define the implementation of a method of one of the server interfaces by the component
    - ie. the *print()* method seen before
  - **CALLMINE** : to invoke one of the methods present in the server interfaces
  - **REQUIRED** : to access to client interfaces
  - **DATA** : to access to instance data
  - **ATTRIBUTES** : to access to attributes of the component
  - **CALL** : to invoke a method on an interface, passing the arguments
- Documentation online
  - http://fractal.objectweb.org/cecilia-site/current/toolchain-parent/cecilia-adl/thinkmc.html

Cecilia: summary

- Environment to program C based software components
- Suitable for general purpose or embedded development
- Based on the open and flexible Fractal model
- ADL and associated toolchain also open and extensible

Cecilia tools & libraries

Tools / Libraries

- `Maven packaging`
- `Makefile + ceciliac`
- cecilia-baselib
- Minus
- Comete

Maven packaging

- Cecilia libraries and application can be built and packaged by means of Maven
- This allows to have simple automatic and transitive dependency resolution for your application
  - => *"My application needs cecilia-baselib-xxx"*
  - Maven will fetch it for you, and the Cecilia toolchain will seamlessy be able to retrieve that library when compiling your application
  - You don't even need to download and install the Cecilia toolchain itself, Maven will take care of it if you don't have it yet
- You may as well package and deploy your own libraries for simpler reusage and distribution
- Documentation
  - http://fractal.ow2.org/cecilia-site/current/maven-parent/index.html

Makefile / ceciliac

- **For allergics to Maven or old school programmers :-)**
- **`ceciliac`**
  - Is the command line interface tool to compile Cecilia applications
  - Basically a wrapper to the Cecilia toolchain
- Documentation
  - http://fractal.ow2.org/cecilia-site/2.1.0/toolchain-parent/cecilia-adl/ceciliac

cecilia-baselib

- A minimalistic set of components interfaces (expressed in the Cecilia IDL language) and corresponding C component implementations files to provide base functionalities that you may require, such as:
- `boot.api.Main` interface
  - every Cecilia application is required to provide an entry point interface of this kind. (See for instance the HelloWorld example)
- `memory.api.Allocator` interface
  - generic interface for memory allocation and deallocation
  - currently ships with Unix implementation
- Documentation online
  - http://fractal.ow2.org/cecilia-site/current/runtime/cecilia-baselib/

Minus

- It is a component library for building systems and middlewares
  - File system
  - Frame buffer
  - Threading
  - ...
- Extensions to the Cecilia toolchain to provide stub/skeleton generations for various protocols
- Documentation online
  - http://fractal.ow2.org/minus-site/current/minus-site/
  - http://fractal.ow2.org/minus-site/current/root/minus-parent/index.html

Comete

- It is is a component-based middleware that allows to dynamically deploy and bind components on MPSoCs and distributed architectures
- Its component-based architecture eases its porting on various systems and platforms
- It currently supports 3 platforms: Posix, the Traviata system and xSTream
- Documentation
  - http://fractal.ow2.org/minus-site/current/comete/
  - http://fractal.ow2.org/minus-site/current/comete/Comete_userManual.pdf
  - http://fractal.ow2.org/minus-site/current/comete/Comete_tutorial.pdf
  - http://fractal.ow2.org/minus-site/current/comete/CBSTxST-0_7.pdf