THE SEAL COMPONENT MODEL

R. Chytracek†, CERN, Geneva, Switzerland
P. Mato, CERN, Geneva, Switzerland
L. Tuura, Northeastern University, Boston, USA

Abstract

This paper describes the component model that has been developed in the context of the LCG/SEAL [6] project. This component model is an attempt to handle the increasing complexity in the current data processing applications of LHC experiments. In addition, it should facilitate the software re-use by the integration of software components from LCG [5] and non-LCG into the experiment's applications. The component model provides the basic mechanisms and base classes that facilitate the decomposition of the whole C++ object-oriented application into a number of run-time pluggable software modules with well defined generic behavior, inter-component interaction protocols, run-time configuration and user customization. This new development is based on the ideas and practical experiences of the various software frameworks in use by the different LHC experiments for several years. The design and implementation choices will be described and the practical experiences and difficulties in adopting this model to existing experiment software systems will be outlined.

INTRODUCTION

LCG/SEAL project aims to provide software infrastructure, basic frameworks, libraries and tools as a common base to build software for experiments in the LHC era. The process involves selection and collecting of foundation and utility libraries and their adaptation to obtain a coherent set in order to support development of experiments' software. In addition to that the SEAL project is active in the area of developing a set of basic **framework services** to facilitate the integration of LCG and non-LCG software.

Scope of the SEAL project

There are three major domains of the SEAL project activities:

- Foundation class libraries
- Mathematical libraries
- Basic framework services

The set of foundation libraries includes the commonly used libraries like STL, Boost, CLHEP then utility libraries, system isolation libraries and domain specific libraries. The common rule applied here for selection process was to re-use as much as possible of the already existing software.

The work on mathematical libraries aims to provide new generation of object-oriented libraries for High Energy Physics (HEP) community following the same principle as foundation libraries.

The domain of basic framework services covers **component model**, reflection, **plug-in management**, incident (event) management, distributed computing, grid services and scripting.

†This work received support from Particle Physics and Astronomy Research Council, Swindon, UK

COMPONENT MODEL OVERVIEW

One of the important aspects of software development is to keep complexity of software products at reasonable level in order to minimize maintenance and integration costs. The modern software systems try to resolve this problem by using component based programming techniques.

This trend is followed by the SEAL project by providing set of software building blocks (**components** and **services**) which expose well defined interfaces and must be configurable by an end-user if required (**properties**). Component based applications should be easily assembled using a generic infrastructure (**contexts**, **scopes**). The process of component assembly has to be enabled in both at compile-time (**static binding**) and at run-time (**dynamic binding**) based on plug-ins architecture, which simplifies deployment of pre-built components. The guiding principle is to keep the whole framework as simple as possible to allow for short learning curve and at the same time giving robustness and flexibility to developers.

Re-usability

Components should be operating outside their original environment where they were created and this without any need to rebuild them. In the cases where their default behaviour is not sufficient, the components should enable re-configuration to fit the needs of the target running environment.

In a more demanding environment with specific needs not foreseen by the component developers they should be able to customize the components by either overriding some parts of the implementation or providing alternative implementations following the component's interfaces.

There are applications that need dynamic reconfiguration according to some external input which requires different components implementations. Such case requires so-called **software hot-swap** functionality,

which means, run-time replacement of existing component with a compatible one [1].

COMPONENTS À LA SEAL

Component programming in the SEAL project is enabled by a set of base classes, interfaces and protocols needed to build complex software systems. This is what we call the *component model*. These base classes provide standard functionality required for component programming like identification of components, loading of component libraries and their instantiation in a hosting environment, and finally the configuration of them according to the user or developer needs. The users of the SEAL component model do not have to worry about memory management as this is built into the SEAL component base classes, thus removing burden of explicit object memory management code.

There are two basic kinds of component provided to the developers, the *Component* and *Service* classes.

Component

Component provides basic component functionality described above. In order to build a new component one has to inherit from this class and implement the required functionality. Instances of the *Component* class and its derivatives are supposed to be relatively simple "gadgets" exposing usually a single interface in a component based system performing a single well defined function.

Service

In cases where a single component can not do well its job, a group of components may accomplish the mission in a collaborative way. For this purpose the *Service* base class has been introduced to make instances of whole sub-systems by grouping various sub-components or services in one scoped collaboration. Services may expose multiple interfaces reflecting their internal structure and responsibilities.

CONTEXTUAL COMPOSITION

Components can't live in a void space as they need an environment where to live. All-in-one component able to perform all possible tasks is impractical, so there is a clear need to have a mechanism which allows locating other component instances in a component system to exploit their functionality in a collaborative way. Components are often used by several other components at the same time, therefore it is very important that they don't silently disappear causing very likely a failure of the whole application. In addition to this, the component references or component handles are not always available to the developers, especially in the cases where they are dynamically loaded during run-time by other components. Automatic component life-time management and ownership is thus very important aspect in component programming.

All these issues are tackled in the modern component frameworks (J2EE, COM+, .NET, CORBA Component Model) by using a *context* based *component composition* approach [7]. The SEAL project has adopted this approach as well and provides *Context* class which acts as universal component management and wiring system. The SEAL component model follows three basic principles:

- Each component is instantiated into exactly one context
- A component or service must be available via context when it is needed
- Peer-to-peer component communication is encouraged but avoiding prior knowledge of the two peers by use of a third party (observer-notification pattern).

Context

The *Context* class allows organizing instantiated components in a hierarchical tree-like structure (see Figure 1). At the same time it takes over the components' ownership and their life-time management. Having the knowledge about each component in the system, the *Context* class allows performing searches throughout its hierarchy and thus locating other components in the system. Each component instance is associated to a single *Context*, and it gets through it the access to the rest of the application components.

The implementation of contextual composition in the SEAL component model is inspired by Iguana [4] and Gaudi [3] projects.

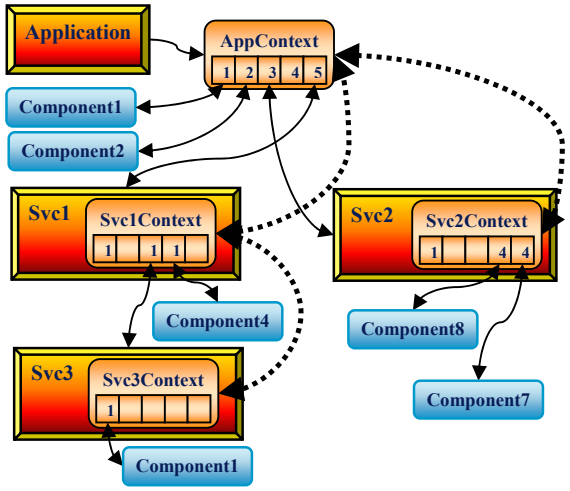

Figure 1: Snapshot showing hierarchical component organization with help of *Context* objects; the dotted arrows represent internal contexts' links of the context tree-like structure.

COMPONENT CONFIGURATION

Components in SEAL can declare their properties to enable their re-configuration if needed. These properties can be inspected, set or updated by the other components in the SEAL component system. The properties can be defined as:

- Template class *Property* instances of any C++ type that has defined the stream operator <<
- References to data members of any C++ type that has defined the stream operator <<

All properties have their associated name and description. The names of Properties are bound to a scope or a namespace to allow distinguishing properties of two or more instances of the same component in different contexts. The scopes of property names are defined by *Services*.

In order to keep the consistent state of a component, when its property has been set or updated by some other component, there is the possibility to associate a call-back function object that is called when this happens. All the properties are stored in the *PropertyCatalogue*.

COMPONENT LOOK-UP

Collaboration of components during run-time allows designing proper functional decomposition spread over a number of distinct functional units. In order to let the components collaborate some component discovery mechanism must be in place.

In the SEAL component model this mechanism is provided by the *Context* class. The SEAL components do not have to know the application topology in order to locate other components. Any SEAL component or service knows about the context where it lives and thus is able to delegate the search task to it. Any component can be located by specifying of either its type or combination of type and key associated to a particular component instance. The look-up by type is performed throughout the context hierarchy, starting from local context and going up in the hierarchy, by comparing the components type ids and stops whenever a component of the given type is found. Similarly, the look-up by key is performed in combination with look-up by type. Unlike the previous look-up method this search does not stop immediately when a component of given type is found but is trying to match component's key value as well in order to satisfy both search criteria.

COMPONENT CUSTOMIZATION

Full component customization can be done by the component model user or developer in the cases where provided component configuration is not sufficient or missing. In this situation one may need to extend the existing implementation or provide an alternative implementation(s).

Customization by inheritance

This is the standard method used by object-oriented systems in general. This way a developer can override behaviour of an existing component. It is not recommended to change the interfaces of the *Component* or *Services* classes.

Alternative implementation

This method can be used to extend existing component model system by implementing a new component with the same interface(s) of an existing one. In this way one can provide a compatible [1] component implementation and replace an existing one.

Adopting foreign components

A framework developer might want to use an existing component that is not a SEAL component (i.e. does not inherit from the SEAL component base class). It is possible to apply the template class *ComponentWrapper*, which allows injecting this foreign component into SEAL component hierarchy. Once wrapped, such a component can be located by existing SEAL components. The main difference with respect to SEAL components is that life-time management for such a component is not done by the *Context* class. The developer is still responsible for the foreign component instance.

PLUG-IN ARCHITECTURE

SEAL component model works hand in hand with SEAL plug-in management facilities. The SEAL *PluginManager* can perform dynamic loading of a user defined class if a proper factory class is provided.

For SEAL components this has been simplified by providing a generic *ComponentFactory* class. In this way any component in SEAL can be made a plug-in object by writing a single line of code.

SEAL components are usually shipped as so-called SEAL modules. These modules are dynamically loadable shared libraries. One can bundle multiple component plug-ins inside a single SEAL module.

Component loader

The SEAL component model simplifies component loading by providing utility class *ComponentLoader*. This class is able to load any SEAL component plug-in either one at a time or whole bunch of components by specifying their component key prefix. Of course, one can specify the target context where the component has to be instantiated after being successfully loaded.

SEAL SERVICES

SEAL has implemented a set of basic framework services using the SEAL component model described above.

Application

This is a utility service which allows bootstrapping an application which uses the SEAL components. This service defines the top-level application context and allows defining the initial set of components to be automatically loaded at the application start-up.

Message Service

This service deals with the application logging needs. It performs composition of messages coming from different

application components, filtering of the messages according the component defined output level and has a configurable reporting facility.

Configuration Service

The main role of this service is to manage properties of the components and to load them from property configurations. The properties can be supplied by the user in a form of configuration files. The SEAL developers have foreseen multiple configuration back-ends. The current (default) configuration file format is extended Gaudi style options file. More back-ends will come in the future like .INI style options, XML style, etc.

Dictionary Service

Reflection capabilities for C++ based applications require loading of C++ dictionary libraries. This service has been provided in order to avoid hard-wiring the knowledge of what dictionary library holds which class's reflection records inside the application code. The main role of this component is to enable automatic discovery and loading of the corresponding dictionary library by just providing a C++ type name.

ADOPTION AND INTEGRATION OF SEAL COMPONENT MODEL

The component model technology in SEAL has so far not been used in its full extent by LCG clients. One of the main reasons for that is the fact that all the LHC experiments, the main LCG clients, have already developed their own frameworks, which are regularly used in production. This is the most important factor in the rather slower adoption of the SEAL component model so far.

Another important SEAL client, the LCG/POOL [4] persistency framework, has been more active in adopting the latest facilities provided by SEAL project including the component model but due to the same reason as above the use of all component model features has been limited.

The situation has changed with the latest new developments in the POOL project dealing with the Relation Abstraction Layer (RAL). RAL defines a generic interface for accessing relational databases. This abstraction requires loading of plug-ins, specific for a database back-end, triggered by user supplied connection strings that identify the data sources. Due to this, the RAL based POOL application, can decide only at run-time what plug-ins must be loaded and therefore a strong support for run-time re-configuration is needed.

Use-case: POOL RAL and ODBC plug-in

Among other RAL plug-ins, the plug-in for ODBC based access to relational databases is a bit special. It is the generic and the de facto standard API for accessing relational databases. Because of the POOL convention in the connection strings, it is not possible to use standard DSN based connections forcing us to use the so called DSN-less ODBC connection mechanism. This requires that POOL RAL ODBC access is initialized in multiple stages.

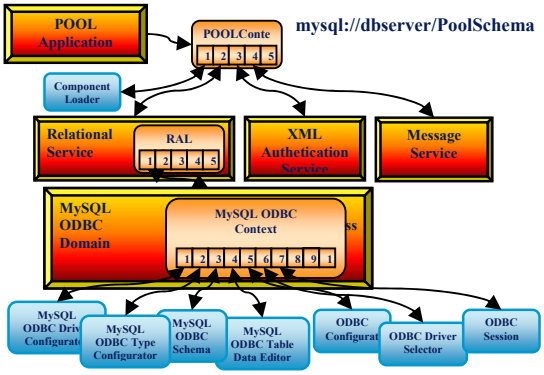

Figure 3: Component lay-out of the POOL RAL ODBC plug-in

Some of the features of the POOL RAL interfaces are impossible to implement in a generic way, even via the ODBC generic API. Most of the cases are related to database schema management, type conversions, index and table manipulations that require database back-end specific handling.

CONCLUSIONS

The implementation of the SEAL component model can be considered complete enough for adoption and/or integration into the client frameworks and applications. It provides the necessary functionality to allow development of component based applications.

The first larger adoption of the SEAL component model is happening in LCG/POOL project for implementation of POOL RAL layer. Further integration is expected in LHCb and ATLAS experiment software when these adopt the SEAL component model into Gaudi framework.

No new major developments are planned of SEAL component model and the development of additional common framework services unless there is an explicit request for a new functionality from the experiments. The only outstanding area in the SEAL component model is the component configuration which needs to be extended by broader support for various property file formats.

REFERENCES

[1] Compatible means the same interface(s)
[2] http://cern.ch/Gaudi/welcome.html
[3] http://cern.ch/iguana
[4] http://pool.cern.ch
[5] http://cern.ch/LCG
[6] http://seal.cern.ch
[7] C. Szyperski, D. Gruntz, S. Murer, "Component Software - Beyond Object-Oriented Programming", Addison-Wesley, 2002, ISBN 0-201-74572-0