Proposal for the Migration of the SEAL Functionality

Introduction

The functionality that is left from the SEAL project is the following:

- **Foundation**, which includes the packages: SealBase, SealIOTools, SealZip and SealUtil.
- **PluginManager** and related utility programs (SealPluginDump, etc.)
- **Component model**, which includes the packages: SealKernel and SealServices.
- **MathLibs**, which still includes FML (fit and minimization) package

This document proposes a number of tasks and actions with the goal to empty the SEAL project from its functionality (i.e. remove all the packages that are left) and be discontinued in the future when all the functionality used by the experiments is taken from other sources. The timescale of this process is not fixed but it should not be extended indefinitely.

Known usages of the different packages

Foundation

These packages are mainly used by CMS directly, and also at some level by the Persistency Framework projects (CORAL, COOL, POOL). Indirectly is also used by LHCb and ATLAS.

Plugin Manager

The SEAL Plugin manager is used by the Persistency Framework project and CMS. The new CMS framework CMSSW relies heavily on it.

Component Model

The two packages of the component model are used exclusively by the Persistency Framework projects. No use in CMS has been reported. Recently, some modifications had to be done to better support multi-threading.

Proposal for the Component Model migration

It is assumed in this proposal that the SEAL component model is not used directly by any of the LHC experiments. In case there still some use of the component model in CMS software it could be due to the fact that some old prototypes have not completely migrated the new software and this should be done.

The following actions are proposed:

1) The SEAL packages SealKernel and SealServices will be moved to the CORAL project. The SealKernel package contains a number of base classes that defines the model (Context, Component, Service, RefCounter, ComponentLoader, etc.) and a number of common basic interfaces (IMessageSvc, IConfiguratlonService, IReporter, etc). The SealServices package includes the implementation of some common services. The Persistency framework packages use some of these services, but probably not all of them.
Therefore, the migration of SealServices will be done only for the services required.

2) The dependency on other SEAL Foundation classes by the component model is very limited (e.g. SealBase package) but not null. For example, the DebugAids, the Error class, etc. are used in the component model. We believe that these dependencies are not fundamental and can easily be removed. A much stronger dependency is to the PluginManager (i.e. the ComponentLoader service) and a solution will have to be found (see later). If no major re-design of the component model is undertaken, the migration of these packages to CORAL should not pose big problems and could be done rather quickly in the order of a week or two.

3) The component model migrated to CORAL will be for internal use of the Persistency Framework packages. The migrated component model is not intended to be made available to end-users. This will simplify the migration process and it will allow possible redesigns of simplifications of some parts. In addition, it will not add maintenance load to the Persistency projects that are suffering currently from scarcity of resources.

Proposal for the Plugin Manager Migration

For the migration plugin management functionality we have discussed two possible options. Since the SEAL plugin manager is heavily used by CMS, it will be difficult to migrate it in any different form than the current one. Doing this would provoke substantial changes in the CMS code, which is not very convenient. Therefore, for the time being we could leave the SEAL plugin manager 'as is' in SEAL until the moment is only used by CMS and then negotiate a migration of it to the experiment’s code base. For what concerns the use of the plugin manager in the Persistency Framework, the two options are:

a) Re-implement the 'ComponentLoader' of the component model with a set of "hardwired" relations between Persistency components and the name of the library that needs to be loaded and the factories to be located in them. This solution is very pragmatic and will remove any dependency to SEAL functionality and is fairly easy to implement. However, it will imply to modify this mapping (plugin<->library<->factory) every time a new plugin is defined in the Persistency framework.

b) The second option is to use the new plugin factory management package (PluginSvc) that has been developed in the context of GAUDI. This package uses Reflex to manage and to invoke factory methods to instantiate plugins using the information from 'rootmap' files that are generated automatically as part of the build process. The factory methods are defined by simple C macros in the user code. There is no need for generating dictionaries for the classes to become plugins. This package depends exclusively on Reflex and on the standard C++ library. See appendix for the use of this package. An agreement has been discussed to add this functionality to the Reflex package in ROOT. This will simplify the packaging and distribution of this functionality and facilitate the use of it by the Persistency framework projects.

Proposal for the Foundation Migration

Assuming that the component model and plugin management has been worked out as is proposed by the previous actions, then experiments and projects would not have a dependency to the SEAL foundation packages except for CMS that will still use directly some parts of it. Therefore, what is proposed it to move the remaining packages: SealBase, SealIOTools and SealZip or portions of them to the CMS code base.
Appendix A – GAUDI Plugin Service

This package has been developed to enhance and simplify some aspects of the GAUDI framework. The main goals have been:

- Remove the need for the property “ApplicationMgr.Dlls”. Component libraries could be loaded on demand
- Simplification of the code. Replace existing factories (AlgFactory, ToolFactory, etc.) with a single method, so many classes can be removed from Gaudi.
- Compatibility with other plugins and dictionary systems since they are based also on roopmap files
- Dependent exclusively on the Reflex package, such that can be added into ROOT/reflex package
- Possible replacement for the SEAL plugin manager that could be of interest for CORAL, POOL, COOL, etc.

Using the package

There is not predefined model on what a plugin/component can be. Any class can be a plugin. The plugin factory is declared in the user code with the exact signature of the constructor and the type returned (base class or interface) by the factory.

```c++
class MyClass : public ICommon {
    MyClass(int, ISvc*);
    ...
};
```

```c++
PLUGINSVC_FACTORY(MyClass, ICommon*(int, ISvc*));
PLUGINSVC_FACTORY_WITH_ID(MyClass, 666, ICommon*(int, ISvc*));
/* implementation */
...
```

The rootmap file, which is a text file containing the association between the plugins and the libraries that implements them, is generated automatically at build time with the help of the genmap utility program. This program loads each library and discovers what plugins it contain.

The plugin can be easily instantiated in the user code by using the class name or an ID class with strong type checking on the constructor arguments. An ID class can be any class that defined operator==( ) and ostream& operator<<( ). The library containing the plugin will be loaded if needed.

```c++
ISvc* svc = ...
ICommon* myc;
myc = PluginSvc::create<ICommon*>("MyClass", 10, svc);
// or PluginSvc::createWithId<ICommon*>(666, 10, svc);
if ( myc ) {
    myc->doSomething();
}
```

Implementation

The package is almost standalone with a exclusive dependency to Reflex. It is rather small and can be added into Reflex practically without increasing the size of it.
Use in Gaudi

The basic use on Gaudi is by replacing the existing C macros with the new macros. This should make transition almost backward compatible.

```cpp
#define DECLARE_ALGORITHM_FACTORY(x) \\    PLUGINSVC_FACTORY(x, IAlgorithm*(std::string, ISvcLocator*))

#define DECLARE_SERVICE_FACTORY(x) \\    PLUGINSVC_FACTORY(x, IService*(std::string, ISvcLocator*))

#define DECLARE_TOOL_FACTORY(x) \\    PLUGINSVC_FACTORY(x, IAlgTool*(std::string, std::string, ISvcLocator*))

#define DECLARE_CONVERTER_FACTORY(x) \\    PLUGINSVC_FACTORY_WITH_ID(x, CnvID(x::storageType(), x::classID()), IConverter*(ISvcLocator*))
```