A COMPONENT FRAMEWORK FOR DEVELOPING SCA COMPONENTS

Dr. Stefan Burkhardt
(Rohde & Schwarz, Munich, Germany, stefan.burkhardt@rsd.rohde-schwarz.com);
Thomas Bleichner
(Rohde & Schwarz, Munich, Germany, thomas.bleichner@rsd.rohde-schwarz.com)

ABSTRACT

Developing SCA resources and devices for Software Defined Radios requires to take care of a large number of requirements regarding the components as such (SCA resources and devices). Code generators producing the framework code often lead to a large number of source code lines, many of which are duplicated in many resources.

In our approach, we are developing a set of standardized building blocks which implement the given SCA interfaces as well as data streaming and control interfaces from the API supplement. Also, internal interfaces which are required for the integration of waveform algorithms are provided. In order to develop an SCA component employing the proposed set of building blocks, the developers have to customize the internal interface of the components and select the corresponding building blocks for the external interfaces from the set. Besides this, the developer merely has to define names and types of the component's properties and the algorithms inside the component. If new interfaces are required, templates are provided which allow to include these interfaces into the set.

Consequently, every SCA component (resource or device) can be built from these building blocks. Only little additional code is necessary to connect the components to the actual assembly.

In contrast to currently existing code generation technologies we do not just generate the code from templates. We provide a resource and device framework in the form of libraries, which cover major common functionality of the components.

In the next step, we plan to extend the approach by incorporating our component framework into an existing SCA tool, so that it automatically generates the connecting code and makes efficient use of the component framework.

1. INTRODUCTION

Developing a Software Defined Radio which incorporates SCA components, i.e. SCA resources and SCA devices, usually requires to consider a large number of requirements for the components stated by the SCA. Developing a component from scratch leads to a significant expense for design, implementation and test.

Existing code generators are able to produce the framework code but also often lead to a large number of lines of source code. Many of these lines are duplicated in many resources.

For example, the realization of an SCA component requires to implement all methods which are defined in the interface. Figure 1 shows the core framework Resource interface for illustration of this point. All methods must be filled with an implementation. Much of this functionality is not component specific and hence also occurs in many other components. By defining a suitable base class, the amount of additional code that has to be written explicitly can be strongly reduced (see Figure 2).

Within the proposed component framework, the functionality which is common to all components is implemented in base classes and provided to the developer as set of software building blocks. The usage of these building blocks leads to a significant reduction of the effort for component creation and testing.

The software architecture of the component framework allows for an easy extension and adaptation to new requirements without the need to change the core component implementation.

2. THE COMPONENT FRAMEWORK

Contents

The component framework includes a number of base libraries for the development of SCA components as well as support to test the resulting components.

Base Libraries

The current version of the base libraries is designed to speed up the component development process. This is achieved by providing a set of base classes which free the programmer from coding common behaviour multiple times. This

```
class ResourceImpl : public POA_CF::Resource
{
public:

    // Construction of Resource: Creation of worker,
    // Creation of ports, creation of component,
    ResourceImpl();

    // Destruction
    virtual ~ResourceImpl();

    // Implement return of identifer
    virtual char *identifier();

    // Implement operations for start
    virtual void start();

    // Implement operations for stop
    virtual void stop();

    // Implement initialisation of component,
    // activation at POA etc.
    virtual void initialize();

    // Implement release of component,
    // deactivation, shutdown etc.
    virtual void releaseObject();

    // Implement runTest handling beside the actual tests
    virtual void runTest( ::CORBA::ULong testid,
                         |::CF::Properties &testValues);

    // Implement handling for configure of properties
    virtual void configure(const::CF::Properties &configProperties);

    // Implement handling for query of properties
    virtual void query(::CF::Properties &configProperties);

    // Implement request for Ports
    virtual CORBA::Object_ptr getPort(const char *name);
};
```

Figure 1: Implementation of the Resource interface without base component

reduction in the amount of coding directly leads to a reduction in coding errors.

Today, the software of the libraries includes:

- An encapsulated abstraction of the POSIX process and thread API.
- Standard component building blocks for the SCA Resource and Device interfaces.
- An implementation of the general resource and device behavior which includes:
    - the registration for resources and devices at the naming service and at the device manager respectively.
    - the start of the component core implementation.
    - the activation of the POA (portable object adapter).
    - and the correct shutdown after a releaseObject() call.
- An implementation of frequently required ports, including log and event port, and several data stream ports.
- Support for the creation of new uses- and provides-ports with interfaces using the SCA building blocks.

```
class ResourceImpl
: public RsSca::ResourceBase
{

public:

    // Creation of ports and worker
    ResourceImpl();

    // Destruction of component
    virtual ~ResourceImpl();
};
```

Figure 2: Implementation of the Resource interface with base components Figure

- Support for the creation of user-defined uses- and provides-ports which implement a freely defined IDL. In this context, a freely defined IDL is an interface definition that neither consists of nor contains SCA building blocks. Such IDLs may stem from the import of already existing interfaces, for example.

The implementation of the base libraries also takes care of SCA specific requirements of the components which comprise:

- Restriction to the SCA AEP for SCA resources.
- the SCA requirements for the base application interfaces and the device interface.

Hull and worker of a component

The libraries follow the notion that each SCA component consists of two parts: the component hull and the worker.

The component hull includes the overall component management, i.e. starting and stopping the component, managing ports, taking care of configuration and lifecycle etc. The definition of the hull is fully supported and provided by the framework.

In contrast, the worker only takes care of the implemented algorithm as well as the definition of the component specific properties. SCA properties like, for example, the property PRODUCER_LOG_LEVEL for a log producer are automatically inserted when using a log port inside the component. For the definition of the worker, the framework provides at least a base class for the worker. Furthermore, property classes are provided which facilitate the definition of various property types (simple properties, simple sequence properties, struct properties and struct sequence properties) by defining names, access rights and value ranges. Since the base component libraries adhere to the SCA AEP and also cover a large part of the component implementation, programmers only have to concentrate on the remaining parts to be SCA AEP compliant.

As a prerequisite for component development, the existence of the definition of the component type, the ports and port types as well as the component properties (type, name, access right, value range) is assumed. This definition is part of the design and leads to the implemented SCA component as well as to its XML descriptors.

Given a component definition, the component development consists of the following steps:

- Selecting the correct component base class. There are different base classes for resource and the various device types.
- Selecting the correct ports. If a required port type does not exist within the framework, the port must be implemented by the developer and can be added to the framework.
- Definition of the worker including the implementation of the component specific properties and the connection between hull and worker.
- The development of devices includes the definition of the capacity properties. These are automatically used by the device's capacity management and state machine.
- At last, the actual worker code has to be implemented. This implementation is supported by the worker encapsulation which provides the function prototypes of the SCA AEP.

The handling of the resource methods start(), stop(), initialize() and releaseObject() is provided by the component framework. If a component requires specific actions during these calls, the worker provides the possibility to implement the desired functionality.

The handling of the device methods allocateCapacity() and deallocateCapacity() as well as the state machines is also provided by the framework. The device's worker only has to define the capacity properties.

Support for user-defined enhancements

From our experience, an user-defined enhancement of the framework usually consists of the definition of a new port IDL and the creation of the corresponding blocks.

As denoted in the previous section, this task is supported by:

- Using a common base class for each port. This base class takes care of the inclusion of the port in the component's internal management structure and the handling of the port in the component framework. By handling we mean the publication of the port name outside the component for the PortSupplier::getPort() method and the return of the correct object references to the CORBA servant. Furthermore, the handling includes correct activation and deactivation of the CORBA servant at the corresponding POA (portable object adapter).
- Using a base class for each uses-port. The base class is itself derived from the common port base class (as described above) which ensures that the uses-port is handled as an external port. Furthermore, the base class implements the CF::Port interfaces and thus handles connection and disconnection, checks the object references for correctness during connections, and handles multiple connections. Also an iteration is implemented with which method calls can be forwarded to all currently active connections. With all these pre-implemented features, the creation of an uses-port just requires the instantiation of the uses-port base class together with the corresponding connection type (CORBA interface pointer).
- Providing a base implementation for the servants which can be created from the SCA building blocks. Hereby, the possible customization of the building block IDL with, for example, different payload and control type is handled. E.g., the SimplePacket building block knows two template parameters: ControlType and DataType. A customization of the same building block with different types is possible and hence different servant implementations are required. The developers of the framework should not assume anything about the customization by the user. Thus, the implemented and provided servant must consider that several types are possible. This leads to a servant implementation in form of a C++ template.

For the creation of user-defined ports only a few steps have to be carried out:

- If the new port IDL only consists of building block IDLs, the corresponding base implementations have to be assembled.
- If the new port IDL consists of existing building blocks as well as user-defined IDL types, the port implementation can be done by choosing the corresponding building block base implementations, and enhancing them by an implementation for the user-defined IDL.

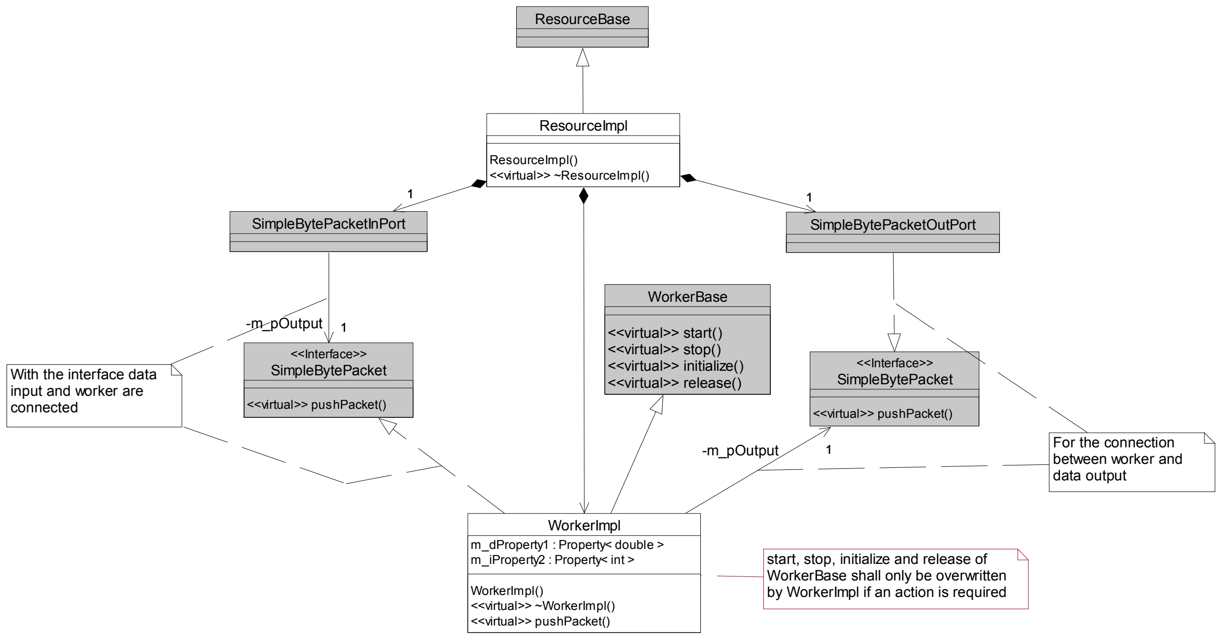


Figure 3: Example design of an SCA resource with base components. Base component parts are with grey background.

- For the case that the IDL is freely defined, the new port shall be assembled from the port base class by implementing the IDL methods.

Support for the assembly controller

Each SCA waveform application contains a special component, the assembly controller. This assembly controller can be one of the functional resources or a separate resource.

The development of the assembly controller is also supported by the framework.

A special port which is a uses-port for the core framework's resource interface is defined. This port seamlessly integrates into the framework, and takes care of the handling and distribution of method calls to the assembly controller's resource interface. The methods start and stop are distributed. Property handling is possible by distributing the properties to the affected components (resources and/or devices). Furthermore, an internal interface for defining a specific mapping between properties and the checking and adaptation of property dependencies between several components is provided.

Support for testing

Together with the component framework, a test framework has been developed.

Each component hull is built upon previously tested building blocks. Possible source of errors can arise from:

- Errors during the implementation of user-defined ports with new interfaces.
- Inconsistencies between the component and its XML descriptors. These inconsistencies may occur between the property types, names and access rights as well as the port types, names and implemented IDL.

For the test against the first error source (errors during implementation of a port with an user-defined IDL), the definition of unit tests is suggested. These unit tests can be derived from the tests of the port base implementation and have to be enhanced by special tests dedicated to this port.

For ensuring the consistency between the component and its XML descriptors, a test suite is provided. This suite reads the XML descriptors, checks the appearance and functionality of ports as well as the appearance, correct type and correct access rights of properties. Furthermore, the correct external behavior of the component is verified.

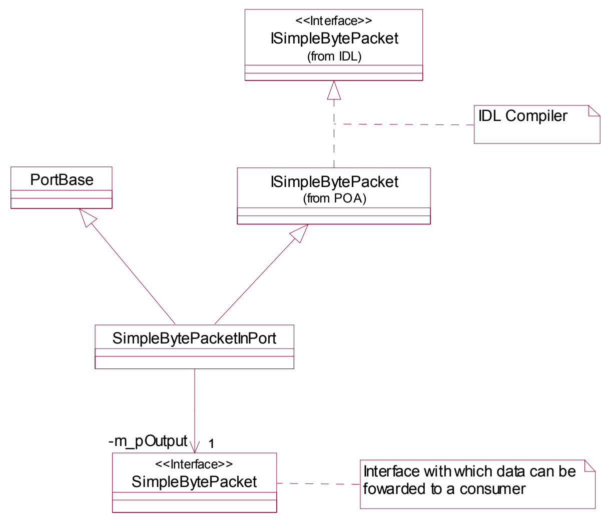


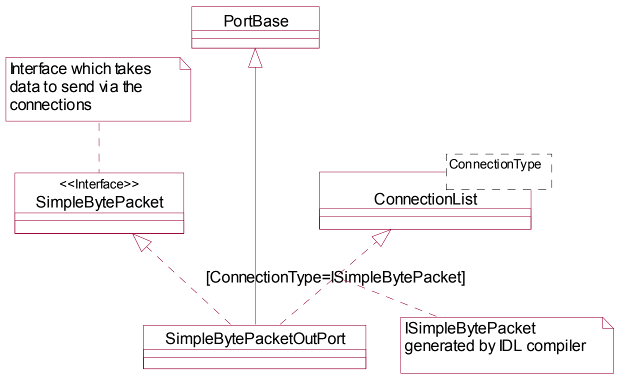
Figure 4: Design of a provides-port

3. RESULTS

The above described framework has been used during the development of a Software Defined Radio platform and a waveform application in a number of components. This section will summarize the benefits of and experience with the framework.

Table 1: Lines of source code for a number of SCA components without functional code.

Component #	1	2	3	4	5	6
Number of ports	2	3	3	10	5	0
Number of properties	1	2	54	13	26	19
User-defined port types	0	0	2	6	4	0
SLOC overall (frame and user-defined port types)	74	81	335	345	292	97
SLOC for user-defined port types	0	0	118	251	180	0
SLOC for hull	**74**	**81**	**217**	**94**	**112**	**97**
SLOC without framework (for comparision)	200	245	-	-	-	-



PortBase
Interface which takes data to send via the connections
<<Interface>>
SimpleBytePacket
ConnectionType
ConnectionList
[ConnectionType=ISimpleBytePacket]
SimpleBytePacketOutPort
ISimpleBytePacket generated by IDL compiler

Figure 5: Design of a uses-port

Example Component

Figure 3 shows an example for the implementation of an SCA resource with the base component framework. Classes with grey background are part of the base component framework. It can be seen that much of the required implementation is already covered by the components provided by the framework. For the implementation, the developer only needs to take care of the worker implementation and the specific resource implementation. In the worker implementation, just the specification of the properties and the implementation of the data processing (here: pushPacket) must be done. The implementation of the specific resource is restricted to the creation of the required ports and the correct connections for the data flow (data input → worker → data output). If worker-specific actions during initialize(), start(), stop() or releaseObject() are required, one will have to add the corresponding methods to the worker overwriting the methods of the class WorkerBase.

Benefit for component development

Because of the building block structure, the development of a standard component hull can be done in little time which is, depending on the number of ports and properties, typically less than two days. Table 1 shows the number of source code lines for components with different numbers of ports and properties. These components do not contain any functional code, only the runnable component hull is measured. For convenience, both the total number of lines of code are shown as well as the number of lines for the frame and the number of lines for user-defined ports. As shown in the table, the resulting C++ code of such a standard component hull usually has a total size of only a few hundred lines of source code which have to be written by the programmer. The term “standard component” describes a component which is built upon already existing port types. As one can see in the table, the implementation of user-defined IDLs leads to additional lines of code.

If it is necessary to define ports with user-defined interfaces, then the time needed to implement the ports for such an interface with a given IDL is usually below half a day. This includes the time for designing and implementing the provides- as well as the uses-port, as shown in Figure 4 and Figure 5 respectively. Additional effort has to be spent if the mapping between port and worker includes the conversion of data types, e.g. between CORBA structures and C++-defined structures.

Table 1 shows also the resulting lines of code for two component hulls in case that no component framework is used. For both components, the number of code lines decrease by a factor of about 2.7-3.0 if the framework is used. However, it has to be noted that individual decrease in the number of source code lines may change from component to component.

Component test

The tests have to be carried out after finalizing the development of the component and the creation of its XML descriptors.

It has been experienced that a major part of the coding errors belongs to spelling errors and wrong types.

After the tests, the correct functionality of the component hull and the consistency between profile and component is ensured. Further tests are necessary after the implementation of the worker. However, since the worker implementation shall not change the component hull, the consistency between component and domain profile can be prooven again with the provided tests.

4. FUTURE DEVELOPMENT

In the future, the component framework shall be integrated into an automatic code generation process which also includes an automatic consistency check between component hull and the XML descriptors of the component. Since consistency errors between component hull and the XML descriptors consume time for correction and mostly the sources are either spelling errors or forgotten and wrong properties or ports, this consistency assurance will lead to a further increase in efficiency. Furthermore, if all ports are implemented, the creation of the component hull is usually a simple step which can be automated.

As a first step, we plan the customization of an existing SCA tool: our framework shall be incorporated in the SCA tool to generate code which effectively uses the component framework. This automatically leads to the consistency between the component and its XML descriptors, since SCA tools generate the descriptors and the component from the same input.

Further steps are necessary to automate the definition of ports with user-defined interfaces, which either consist of building blocks or have a completely new-defined IDL.