# Whitepaper: Gateway Components for Communication between ROS and SmartSoft

Alex Lotz

October 7, 2011

## 1 Introduction

There are several different ways of using *Robot Operating System (ROS)*[1] and SmartSoft components together (resp. to communicate between them). From a technical point of view, one can either port ROS components to SmartSoft or port SmartSoft components into ROS as a stack. Both solutions have the disadvantage to maintain two branches in parallel. Furthermore, to port all components from one robotics framework to another would mean to fundamentally redesign all components from one of the frameworks. This is very time consuming, error prone and thus not practicable.

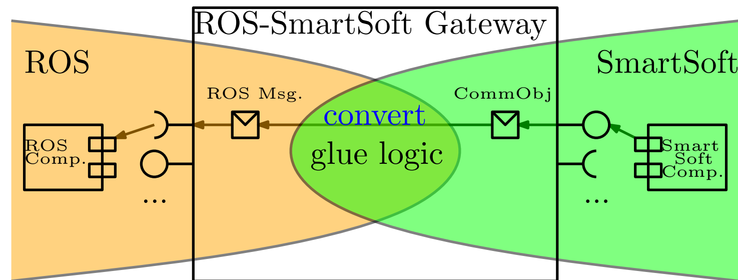


Figure 1: Gateway Component between ROS and SmartSoft

Another approach, to bring the two worlds together, is to build *gateways* (see fig. 1), which allow to communicate between components running in ROS and components in SmartSoft. The advantage of this approach is the full decoupling of semantics and syntax of the two frameworks. The glue between the frameworks is implemented inside the gateway components. The disadvantage of this approach is the additional communication overhead (resp. the jitter) caused by the gateways itself, which however is linear and can be taken into account at system design.

[1] `http://www.ros.org/wiki/`

The gateway approach leads to the following design decisions. First, all components implemented in one of the two frameworks remain untouched and communicate with components from another framework exactly in the same way as they did before. Second, the mapping of the communication details and semantics is implemented inside of specific gateway components. It is more reasonable to implement several gateway components instead of a single gateway implementing all mappings for all potential communication ports. The reasons are to prevent a single point of failure and to provide better scalability and maintainability of gateway solutions.

The solution for the gateway approach leads to the following challenges. First, the two different building processes must be combined in such a way that it becomes possible to build (create and compile) gateway components, which uses both types of communication mechanisms (those from ROS and those from SmartSoft). This issue is described in section 2. The second challenge is to combine the usage and the mapping of the two different types of communication ports inside the gateway components. This issue is described in section 3.

# 2 Combining the building processes of ROS and SmartSoft

As stated before in section 1, the gateway approach decouples the sphere of influence between components created in ROS and components created in SmartSoft. Thus, the only components, which must be created for both frameworks, are the gateway components. To be able to create such gateway components, two technical issues must be solved first.

## 2.1 Technical challenges

The first problem is, that the folder structure in ROS is individual for each of the ROS stacks, whereas in SmartSoft the locations for components, communication objects, utilities, header files and libraries are predefined and stable. This makes it possible to use generic makefiles in SmartSoft, whereas in ROS the locations for header files and libraries are resolved by the tool `rospack`, which in turn uses the bash environment modified by ROS scripts. In particular, the dependencies of a ROS component to other stacks in ROS is defined inside of the `manifest.xml` file. This file is used by cmake to generate corresponding makefiles (e.g. for a g++ compiler). Instead in SmartSoft a more straightforward way is used, where a makefile simply uses the location `$SMART_ROOT/include` as include path and `$SMART_ROOT/lib` as library path.

The second problem is the creation procedure of components in ROS and in SmartSoft. In ROS, a component is created by using the tool `roscreate-pkg`. Thereby the files `manifest.xml` (including the dependencies to particular stacks in ROS) and `CMakeLists.txt` are created. In SmartSoft, a component is best created by using the SmartSoft MDSD Toolchain. Thereby the dependencies are automatically generated by the toolchain.

## 2.2 Solution for the combined building process

In SmartSoft, components and communication objects are separated in corresponding locations:

**Components:** `$SMART_ROOT/src/components`

**Communication Objects:** `$SMART_ROOT/src/interfaceClasses`

Whereas in ROS, a component is found by using the environment variable `ROS_PACKAGE_PATH`. To be able to use a gateway component in ROS and SmartSoft together we create the component inside the components directory of SmartSoft and attach this path to the ROS package path. Thus the first step is to setup the ROS package path to know the location of components in SmartSoft:

```
# cd $HOME
# echo "export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:\
  $SMART_ROOT/src/components" >> .bashrc
```

This step must be done once, after ROS and SmartSoft are first installed on a system. Now for each new gateway we create a new ROS package[2]:

```
# cd $SMART_ROOT/src/components
# roscreate-pkg <GatewayName> roscpp tf [further ROS depends.]
```

The parameter `GatewayName` is individual for each gateway component (e.g. `SmartROSBaseGateway`). In addition to the two packages `roscpp` and `tf` further dependency packages can be specified. This is the minimal definition for a new component in ROS. Now the SmartSoft MDSD Toolchain can be used to create a component body for a SmartSoft component.

After creating a SmartSoft hull for the gateway component, the library dependencies of ROS must be defined. Thereto, a predefined bash script generates a makefile, which contains all compiler and linker flags for the ROS dependencies. Again, in ROS all dependencies are stored inside the `manifest.xml` file of the gateway-component project. The script can be executed as follows:

```
# cd $SMART_ROOT/src/utility/smartROSbridge
# ./generate_ros_dependencies.sh <GatewayName>
```

The script not only generates the top level library dependencies but also the recursive dependencies of these libraries and so on. At the end, the information is stored in the file `ros.mk`, which in turn is stored in the directory:

```
$SMART_ROOT/src/components/<GatewayName>/src
```

Now the ROS specific makefile parameters can be included into the SmartSoft `Makefile` of the corresponding gateway component:

[2] http://www.ros.org/wiki/ROS/Tutorials/CreatingPackage

```
# roscd <GatewayName>
# cd src
# cat ros.mk
# vim Makefile
```

The `Makefile` in SmartSoft provides the variable `USER_CPP_CFLAGS` for user defined *compiler* flags and the variable `USER_CPP_LFLAGS` for user defined *linker* flags:

```
...
USER_CPP_CFLAGS =
USER_CPP_LFLAGS =
...
```

This part can now be extended by an include of ROS specific compiler and linker flags (currently generated by the script above):

```
...
ifneq "$(wildcard ros.mk)" ""
include ros.mk
else
$(error "File 'ros.mk' NOT found! run \
  ./gen_ros_depends.sh in bash console!")
endif

USER_CPP_CFLAGS = $(ROS_CXX_FLAGS)
USER_CPP_LFLAGS = $(ROS_LD_FLAGS)
...
```

That's it, we now can compile the gateway by simply typing:

```
# roscd <GatewayName>
# cd src
# make
```

Seamlessly, the build procedure of the SmartSoft MDSD Toolchain can be used without any further modifications.

# 3 Glue Logic

Once we are able to compile a component consisting communication ports of ROS and SmartSoft the next challenge arises. As a matter of fact, neither the messages in ROS nor the communication objects in SmartSoft are standardized. As a consequence, there is no generic mapping possible even between similar data types. For example, ROS provides the message `Odometry`, which is quite similar to the communication object `BaseState` in SmartSoft. However, there are differences in internal structures, variable names, data types, physical

units (resp. meaning of the value) and expected value ranges. As long as none of the two frameworks follows a standard, the mapping of the communicated data types will remain handwritten and will be highly affected by changes in both frameworks. Nevertheless, one can reduce the proliferation of mapping modifications and their influence on the rest of the system, by outsourcing the mapping to a separate library. This at least decouples modifications in ROS from those in SmartSoft and vice versa.

After implementing the mapping of required communication data types in a separate library a gateway component can be designed. Fortunately, there is a recurring structure in most of the ROS-SmartSoft gateway components. The main parts of it are described in the following.

- SmartSoft Communication Ports of the gateway component can be defined and configured by using the SmartSoft MDSD Toolchain.
- An instance of a ROS Node (required by ROS communication ports) can be defined in `<GatewayName>Core.hh`.
- The best place to define Communication Ports of ROS is the header file `<GatewayName>Core.hh`, initialization can be implemented in the source file `CompHandler.cc`.
- Each gateway needs one specific task (the `ROS_SpinTask`) to handle all ROS middleware events by calling the `spin` method of the local ROS Node.
- Finally, for each `ros::Subscriber` a callback method must be implemented which handles incoming ros messages.