OETCS

openETCS

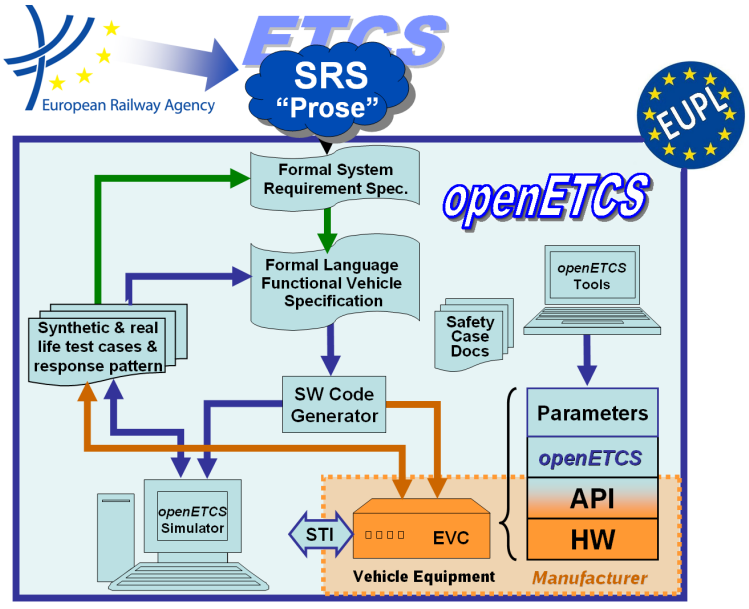
ITEA2 Project
Call 6 11025
2012 – 2015

Work-Package 7: “Toolchain”

Model evaluation report: SystemC model of UNISIG Subset-026-3 3.5 Management of Radio Communication (MoRC)

Roberto Kretschmer, Stefan Rieger

June 2013



Funded by:

Federal Ministry of Education and Research

Région de Bruxelles-Capitale

This page is intentionally left blank

Work-Package 7: “Toolchain”

OETCS
June 2013

Model evaluation report: SystemC model of UNISIG Subset-026-3 3.5 Management of Radio Communication (MoRC)

Roberto Kretschmer, Stefan Rieger

TWT GmbH Science and Innovation
Bernhäuser Straße 40-42
73765 Neuhausen

Model Description

Prepared for openETCS@ITEA2 Project

Table of Contents

Figures and Tables

Figures

Tables

1 Short Introduction to Formalism and Tool

SystemC is a C++ library providing an event-driven simulation interface suitable for electronic system level design. It enables a system designer to simulate concurrent processes. SystemC processes can communicate in a simulated real-time environment, using channels of different data types (all C++ types and user defined types are supported). SystemC supports hardware and software synthesis (with the corresponding tools). SystemC models are executable.

1.1 SystemC Evaluation Models for WP7

Within the context of Work Package 7 we are working on two different models:

- A SystemC model for Subset 026, Section 3.5, “Radio Communication” which is currently developed by TWT
- A SystemC model regarding the braking curves in Subset 026. This model is being developed by the University of Rostock

At the moment (June 2013), the models are still in an incomplete state but at least a basic version of the radio communication model is ready for assessment. We will provide updates in alignment with the review process as soon as we are able to integrate additional features. As the scope of WP7 is *not* to provide a complete system model, we believe that also incomplete models are suitable to get an idea of the modeling language and judge its suitability for application within the project.

Purpose of this Documentation

This documentation shall give a short introduction to SystemC, to our modeling process, which is based on a top-down approach based on SysML and code generation thereof, and the structure of our models.

1.2 Basic Concepts of SystemC

A SystemC model is constructed of so-called *modules*, C++ classes derived from a common base class `sc_module`. Modules may be nested and are interconnected by *channels*. There is a variety of different pre-defined channels, such as signal-channels (behavior as a shared variable or a hardware wire) or FIFO-queues. In addition, it is possible to define custom, complex channels. Modules usually contain *threads* and *event-triggered methods* that are executed in parallel. Threads and methods may register/wait for certain events in the system, such as the transmission of a message via a channel or a value change of a signal.

1.2.1 Defining Modules, Ports and Channels

Figure 1 depicts a simple example structure of a SystemC model. Here we have three modules, where Module A is a sub-module of Module B. Modules B and C are interconnected with two channels. The upper one is a FIFO queue providing a buffered message channel, whereas the lower connection is realized by a signal channel transmitting data directly. For connecting channels, modules provide *ports*. Ports are pre-defined interfaces supporting, e.g., reading or writing operations. Ports are often directed, i.e., there are input and output ports.

To realize the structure from Figure 1 we could declare the modules as follows:

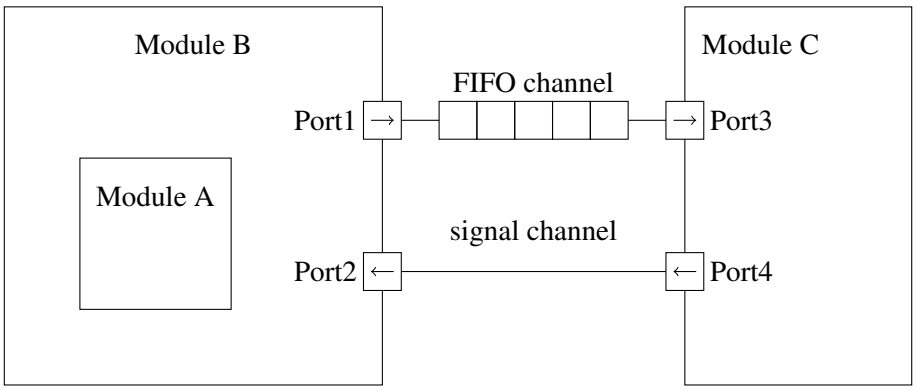


Figure 1. Interconnected SystemC Modules

```
1  SC_MODULE(ModuleA){
2      ...
3  };
4
5  SC_MODULE(ModuleB){
6      sc_port<sc_fifo_out_if<int> > port1;
7      sc_port<sc_signal_in_if<int> > port2;
8
9      ModuleA subsystem;
10     ...
11
12 };
13
14 SC_MODULE(ModuleC){
15     sc_port<sc_fifo_in_if<int> > port3;
16     sc_port<sc_signal_inout_if<int> > port4;
17     ...
18 };
```

As you can see, the ports are declared with the corresponding type (in this case `sc_fifo_out_if`, `sc_fifo_in_if`, `sc_signal_in_if`, and `sc_signal_inout_if`[1]) which are interfaces to the corresponding channels and specified as C++ template parameter. In this example all interfaces have been set up for the message/signal type `int`[2].

1.2.2 Connecting the Modules

Although we have defined all three modules, the interconnections are not defined yet as no module has been instantiated and the connections between the modules have not been established by providing the corresponding channels. This could be done as follows in the `sc_main` function:

```
1  int sc_main(int argc, char* argv[]) {
2      ...
3      // Instantiate modules
4      ModuleB b("ModuleB instance");
5      ModuleC c("ModuleC instance");
6
```

[1]Note that SystemC does not provide an interface that only allows writing to and not reading from signal channels. This makes sense as a read has no effect on the state of the channel (in contrary to FIFO channels). Alternatively one could use `sc_out<int>` instead of `sc_port<sc_signal_inout_if<int> >`. We use the latter to get a unified notation for ports and to simplify code generation from, e.g., SysML.

[2]Note that it is necessary to specify a type here. Of course, also custom types may be used.

```
7       // Instantiate channels
8       sc_fifo<int> fifo;
9       sc_signal<int> signal;
10
11      // Bind ports to channels
12      b.port1(fifo);
13      c.port3(fifo);
14      b.port2(signal);
15      c.port4(signal);
16      ...
17  }
```

1.2.3 Writing to and Reading from Ports and Channels

There is no unified way to access ports or channels as they may be intended for different purposes. E.g, where for a signal a single value suffices one could define a channel type that includes addressing and payload to simulate network protocols. For the predefined channels, such as `sc_signal` and `sc_fifo` methods `write` and `read` are supported to write values to and read values from a channel/port, respectively. The semantics in those cases, however, is different:

While for `sc_signal` writes and reads are executed immediately and do not have any precondition, writes to and reads from `sc_fifo` are blocking if the buffer is full on write or no data is available on read. However, there are also non-blocking versions (`nb_write` and `nb_read`) that terminate immediately and a return value may be used to determine whether actually any data was written/read.

Thus, in our example write or read operations could look as follows:

```
1  // In sc_main using the channels:
2
3  fifo.write(1);              // blocks if FIFO buffer is full
4  int x = signal.read();      // returns immediately
5
6  ...
7
8  // In Module 3 using ports:
9
10 while(true) {               // reads from FIFO ports are often executed
11                             // in a loop as the operation is blocking
12    ...
13    int x = port3->read(); // blocks until data is available
14   ...
15
16 }
17 ...
18 port4->write(0);           // Non-blocking write to a signal port
19 ...
```

In the above code listing it is evident that the access to the ports from the modules is preceded by the -> operator. This is not necessary when directly writing to the channels.

1.2.4 Threads and Event-Triggered Methods

SystemC make heavy use of concurrency and employs an event-based architecture. Events, such as the change of a signal value, can trigger actions. SystemC provides two concepts for this, both of which are realized as member functions within a SystemC module:

Threads (`SC_THREAD`) are versatile processes that can be halted at any moment and any number of times during their execution by using a `wait` statement. A thread is run only once. Often, a thread is meant to run for the whole duration of the simulation. This can be achieved by using an infinite loop. A thread can be suspended for a certain amount of time or until an event occurs. The events that a thread is sensitive to are not necessarily statically defined.

Methods (`SC_METHOD`) will run more than once when they are triggered by an event and cannot be suspended during their execution. In addition, they usually have a static sensitivity list.

Let us look again at our example. In Module C we could define a thread that waits for an incoming integer on Port 3 (FIFO channel) and then writes it out to Port 4 (signal channel). This can be done as follows:

```
SC_MODULE(ModuleC){
    sc_port<sc_fifo_in_if<int> > port3;
    sc_port<sc_signal_inout_if<int> > port4;
    ...

    void mythread(){
        while(true){
            int x = port3->read();    // Read from FIFO
            cout << "Read value " << x << " from Port 3 \n";
            port4->write(x);          // Write to signal
        }
    }

    // Module constructor, here by using the default macro; can
    // also be defined manually if additional parameters are necessary
    SC_CTOR(ModuleC){
        SC_THREAD(mythread);  // Thread initialization
    }
};
```

Equivalently, one could also use the non-blocking read of the FIFO and explicitly use the `wait` statement:

```
    void mythread(){
        while(true){
            wait(port3->data_written_event());
            int x;
            port3->nbread(x);  //Non-blocking read
            cout << "Read value " << x << " from Port 3 \n";
            port4->write(x);
        }
    }
};
```

In Module B we possibly want to output the state of the signal connected to Port 2 when its value changes. This can be accomplished as follows:

```
SC_MODULE(ModuleB){
    sc_port<sc_fifo_out_if<int> > port1;
    sc_port<sc_signal_in_if<int> > port2;
    ...

    void print_change(){
      cout << "Port 2 value changed to " << port2->read() << endl;
    }

    SC_CTOR(ModuleB){
       SC_METHOD(print_change);   // Initialize event-triggered method
        sensitive << port2;       // Make method sensitive to events on port2
    }

};
```

Thus, whenever the signal connected to Port 2 changes its value, the method `print_change` is called. Please note, that in our model of the ETCS radio communication setup we do not use `SC_METHOD`s for a more streamlined approach. Everything that can be done with `SC_METHOD`s can be also done with `SC_THREAD`s.

1.2.5 The SystemC Simulation Scheduler

Here we will later-on add some information about the SystemC simulation scheduler.

2 Modeling Strategy

2.1 Modeling Strategy: SysML

In the following we shortly introduce the basic modeling strategy, which is based on the decision to use SysML as high level modeling language. In line with the openETCS proposal we developed a modeling workflow of radio communication (MoRC) based on Subset 026, Section 3.5, “Radio Communication” of the ETCS standard following a model driven approach, which is based on available open source tools. The main goal of this approach is a step-wise refinement of models of the SUT to executable code. One obvious question in this regards is what underlying engineering process one chooses, i.e. which steps or abstractions levels are suitable for the development of the SUT. On the one hand SystemC allows the definition of system level descriptions on TLM level which can later be enhanced with further details or the selection of a certain Model of Computation respectively. On the other hand even at the TLM level SystemC might be regarded as too detailed for early stages of system development. Here one is rather interested in high level modeling languages such as UML or SysML which are well suited for the overall system concept and for easy communication with stakeholders of diverse expertise. However there is a semantical gap between these high level modeling languages and the executable code. In the literature different approaches are suggested to bridge this gap between UML/SysML and SystemC. These are mainly realized by defining UML profiles to extend SysML by introducing stereotypes that represent SystemC constructs, e.g., [**? ?**]. In our approach we describe use the standard SysML stereotypes to model the MoRC and infer automatically a TLM description in SystemC using code generation. One advantage is that the modeler is not required to learn new language elements. However, since the code generation fixes the semantics of a given SysML model element to a certain SystemC construct the modeler needs to have a basic understanding of this mapping in order to achieve the intended result.

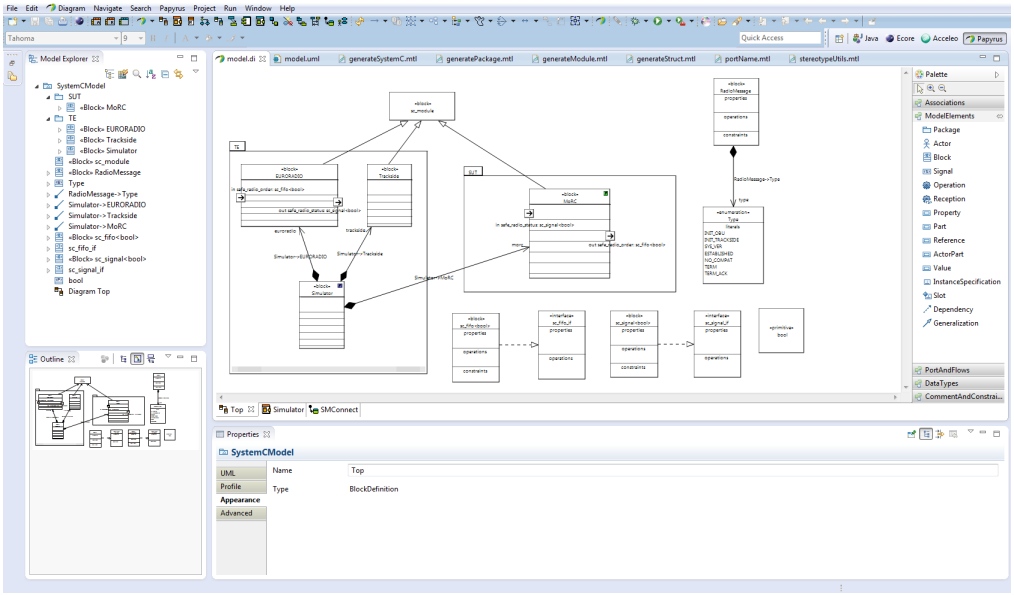


Figure 2. Eclipse IDE with enabled Papyrus plugin, showing the SUT SysML model.

2.1.1 Structural modeling

SysML is models consist of structural and behavior elements. In our workflow we first created a top level block diagram. Blocks represent the fundamental structural unit of the SysML model. To create the SysML model we use the Eclipse modeling framework (Indigo version) with activated Papyrus plug in. Papyrus doesn't support the newest version of SysML 1.3, i.e. full and proxy ports are not defined instead we have used flow ports, which is sufficient for our needs. An example of the Eclipse user interface with activated papyrus perspective is shown in Figure 2.

Figure 3 shows the top level block diagram. This diagram also shows two packages, which are embedded in the top level package, which is called `SystemCModel`.The two nested packages are "'TE"' for the test environment and the SUT package, which contains the actual MoRC model. The use of package allows for a better organization of the overall model and are later mapped to different SystemC namespaces (see below). This way the test environment is clearly separated from all the SUT parts. The TE is needed in order to model how the SUT can be stimulated and eventually simulate it's behavior with the SystemC runtime environment. To distinguish blocks that form the SystemC modules from other block, e.g. primitive and complex data types, we introduced the block `sc_module`and create links of type "'Generalization"' (c.f. Fig. 3). The other blocks are "'RadioMessage"' which is a data type containing of the type of the message and type and interface definitions. The latter are used to specify, i.e. type the flow ports.

The block `Simulator`is the main part of the TE, which is used to connect the instances of the modules and describe the behavior of the TE. At this point we use this block to model the interconnection of the MoRC to the EURORADIO and Trackside model components shown in the internal block definition diagram in Figure 4. Note that this block itself is not derived from `sc_module`,i.e. it is not a SystemC module. The connectors in the internal block definition diagram shows the connection between the ports of the modules that were defined in the top level block diagram. Arrows in the flow ports indicate the direction of the data/signal flow, which can be either in, out, and inout. The type of the port shows the kind of data that is transmitted and the kind of connection: buffer, direct, fifo etc. In this case we connect the ports `safe_radio_status`with a signal and a simple Boolean value is transmitted. For the

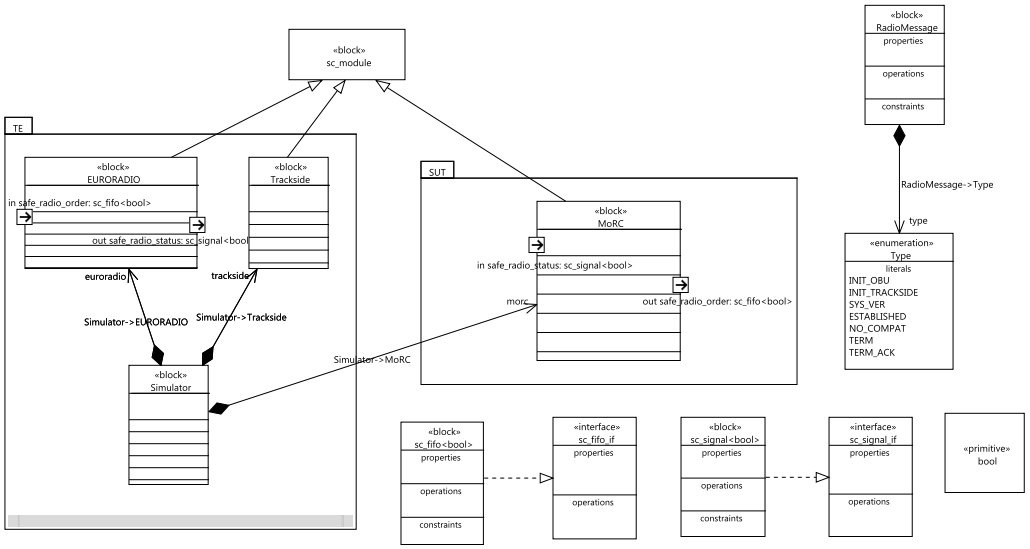


Figure 3. Top level SysML block definition diagram.

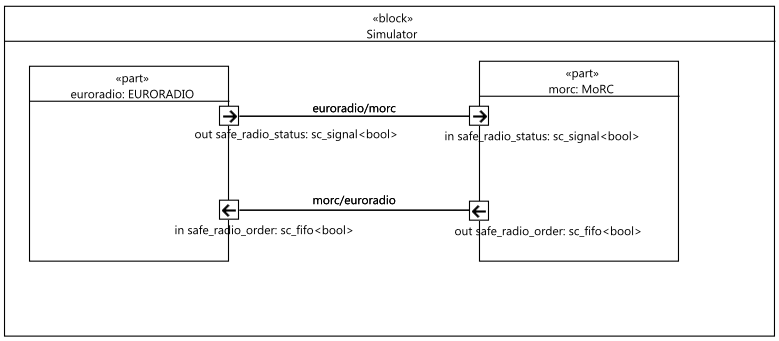


Figure 4. Example of the internal block diagram of the simulator.

`safe_radio_order`we used a FIFO connection instead, which better captures the subset 26 requirements.

To realize the port types we introduce SystemC specific data types via interface and data type blocks (c.f. lower right part in Fig. 3). We added a redefinable template signature to each interface to be able to model the possibility to transmit different data types with the respective interface. The blocks realizing the interfaces (dashed line with arrow in Fig. 3) define a template binding assigning the template to the "‘bool"’ primitive. This template information is not shown in the block diagram and we had to use the model tree editor to realize these links, which is shown in Figure 5. The redefinable template signature includes defines a formal classifier template parameter which is substituted withe actual primitive type `bool`.These described characteristic SystemC model elements might later be added to a dedicated package for easy reuse in other SysML models.

2.1.2 Behavioral modeling

SysML state machine are expecially suitable to model the SystemC event-based behavior. The behavior of the MoRC is modeled with a state machine diagram, which we we will describe here in a upcoming version of this document. The Acceleo-Eclipse plugin supports syntax

hihlighting and code completion for the thus the coding is very convenient. Further features comprise debugging...

2.2 Modeling Strategy: Code Synthesis

In this section we describe the code generation of the SysML model using the Eclipse framework and the Acceleo Plugin. In Eclipse our workflow is organized with two projects. The first, a Papyrus project, contains the the SysML model and the second is an Acceleo project that defines the SystemC code generation. The Acceleo plugin implements the MOF Model To Text Transformation Language (MOFM2T). Using the language one can traverse and query the SysML model tree to access all model elements and their attributes. The SysML/SystemC mapping definition is organized in modules containing templates. These templates are assigned to model elements. The principle of the template is What You Code Is What You Get (WYCIWYG), i.e. all text in the template that is not a MOFM2T language element is put in the transformation output. The language elements contain fundamental control flow constructs, such as loops and if statements as well as variable assignments. In addition it is possible to create files and to extend a template functionality by calling external Java services. For each module we included the SysML and UML profiles, which is needed to declare all types we use in our SysML model. This is achieved by adding the respective URIs when creating a new Acceleo project. The project is linked to our SysML model in the project creation wizard as well.

For the moment our code generator is able to create the structure of the model. For each module a one header and one cpp files is generated. In addition a main.cpp file is created which contains the channel binding according to the SysML model specification. In later version of this document we will describe the translation in more detail and include the transformation of the SysML state machines.

3 Model Overview

3.1 SystemC Model for Subset 026 Radio Communication (TWT)

When creating the SystemC model for the radio communication section of Subset 026, the idea was to have one part representing the system under test (SUT) and one part for the test environment (TE) that provides inputs and evaluates outputs of the SUT. To this end, we defined two C++ *namespaces*, `SUT` and `TE`, each of which may contain several SystemC modules. This lead to the following model structure:

- namespace `SUT`
  - module `MoRC`: The **M**odule **o**f **R**adio **C**ommunication represents the system under test. It receives connection orders from the outside and interacts with the `EURORADIO` module. Includes a state machine `SMConnect` that models the connection setup. (→ morc.h, morc.cpp)
- namespace `TE`
  - module `EURORADIO`: Simulates the behavior of an EURORADIO module setting up the safe radio connection. Failures of safe radio are simulated with an exponential probability distribution. (→ euroradio.h, euroradio.cpp)
  - module `Trackside`: Simulates the trackside (RBC or RIU). (→ trackside.h, trackside.cpp)

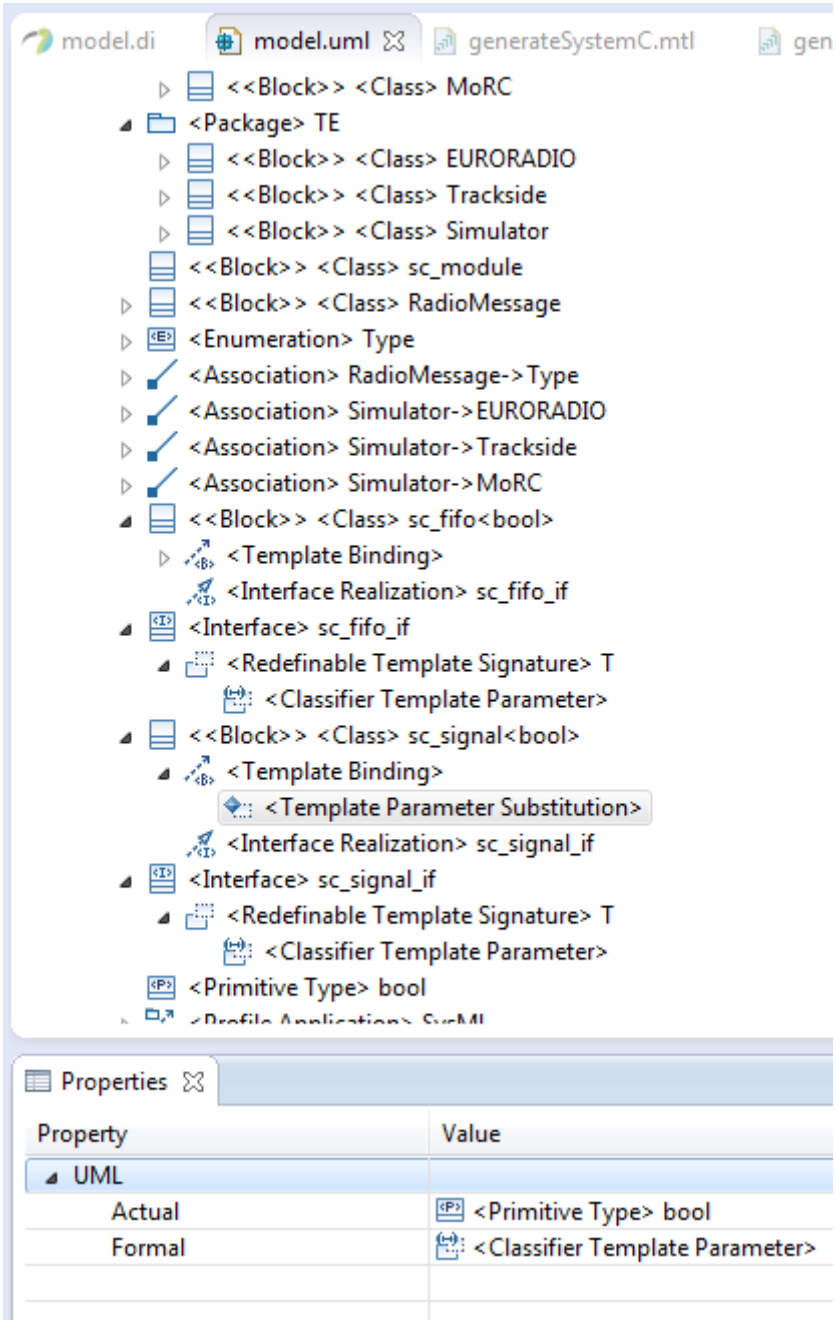


Figure 5. Using the Eclipse model editor to create UML template signatures and bindings

- Global namespace
  - `RadioMessage`: data structure for exchanging messages with the trackside; includes an enum for the message types required by Subset 026, overloaded trace and stream output operators are provided (→ radiomessage.h, radiomessage.cpp)
  - `sc_main`: the program entry point that is used for instantiating all modules, setting up the environment and starting simulation (→ main.cpp)

For a more detailed description of the model please refer to Section 5.2.

3.2 SystemC Model for Braking Curves (URO)

To be filled by Uni Rostock

4 Model Benefits

5 Detailed Model Description

5.1 Technical Realization SysML

SysML mit Eclipse, Interfaces, Templates

5.2 Detailed Description of the SystemC Radio Communication Model (TWT)

In the following we will give a more detailed description of each module of the radio communication model.

5.2.1 MoRC

The Module of Radio Communication (MoRC) represents represents the interface for communication of the ETCS onboard unit (OBU) with the trackside. In the model it is the system under test (SUT). Thus, the MoRC interacts with the EURORADIO module that is responsible for setting up a safe radio communication and via EURORADIO with the trackside. Orders for connection establishment can be triggered by several OBU components, such as the driver module interface (DMI) or the balise transmission module (BTM). In the our model we do not need to worry which component issues an order as they all come in at the same *port*.

In the code we prefix input ports with the letter `i` and output ports with the letter `o`. The MoRC possesses the following ports:

`i_conn_order` Boolean FIFO port that is used by MoRC-external components to trigger the establishment of a connection. If a 1 is received the MoRC will try to establish a connection (if not already established), if a 0 is received it terminates a connection (if applicable).

`o_conn_status` This is a Boolean signal port to indicate to MoRC-external components whether a radio connection has been established.

`o_safe_radio_order` This Boolean FIFO output port is used to communicate with the EURORADIO subsystem. If a message with the content 1 is transmitted, the EURORADIO subsystem shall try to establish a safe radio connection. Upon a 0, a potentially active safe radio connection shall be terminated.

`i_safe_radio_status` Analogous to `o_conn_status`, this signal port is to be used by an EURORADIO subsystem to indicate whether a safe radio connection is established. Upon the establishment of a safe radio connection by EURORADIO, the MoRC will try to connect to an RBC or RIU.

`o_trackside` and `i_trackside` FIFO ports for communicating with the trackside (RBC or RIU). These ports are used for sending and receiving messages of type `RadioMessage` (→ radio_message.h).

In its current version the MoRC module has three threads:

`order_handler()` Upon receiving a connection order via port `i_conn_order` a state machine for setting up the radio connection is instantiated and given control. For more details regarding state machine modeling please refer to Section 5.2.2. Connection termination is not implemented yet.

`safe_radio_handler()` Listens on the port `i_safe_radio_status` to detect the establishment or failure of a safe radio connection. Triggers the *event* `conn_lost_event` if the safe radio connection could not be reestablished within 5 minutes of simulation time.

`conn_status_signal_driver()` This is a set that *drives* the `o_conn_status` signal by listening for the events `conn_established_event` and `conn_lost_event`. Actually this thread is only necessary because in SystemC it is not allowed to write to a signal channel from multiple threads.

5.2.2 Modelling State Machines

In our model the MoRC also contains a SystemC representation of a state machine that is used to model the connection setup. We found that this actually simplifies the control-flow and makes it more readable. In addition this method enables the transformation of SysML state charts to SystemC in a straightforward manner.

The transformation idea is quite simple:

1. Create a C++ function for every state in your state machine, in the following called *state function*. In our model these have been prefixed with `state_` and encaspulated in the subclass `SMConnect`.
2. Each state function may execute a certain actions and before it exits must fix the consecutive state to be executed. This is achieved by assigning to a function pointer the function representing the consecutive state (in our case this function pointer is `SMConnect::next_state`). There may be several alternatives depending on the state of the system. If there is no consecutive state, `NULL` is assigned.
3. Call the function pointer in a loop until its value is `NULL`.

Note: You could achieve the same by introducing a `switch` statement in the main loop and a `case` for each state. But this is less clear in our opinion and increases the amount of code significantly.

The following code excerpt from our model illustrates this process[3]. It shows a state function for the case that a connection order has been received. If the connection status is 1 (connection is established) we are done and assign `NULL` to `next_state`, otherwise we move on to the next state function `state_not_connected_connection_order()`.

```
1 void MoRC::SMConnect::state_connection_order_received() {
2     if (parent.o_conn_status->read())
3         next_state = NULL;
4     else
5         next_state = &SMConnect::state_not_connected_connection_order;
6 }
```

5.2.3 EURORADIO

5.2.4 Trackside

5.2.5 Compilation of the Model

To compile the model we provide a CMake-Makefile (CMakeLists.txt). Assuming that you have CMake and SystemC installed and are using a compiler supporting the C++11 standard you can build the model with the following steps:

1. Set the environment variable `SYSTEMC_HOME` to point to the SystemC installation directory. Alternatively you can edit the file CMakeLists.txt to point to your paths.
2. Create subdirectory "build" and change to it "`mkdir build && cd build`".
3. Execute "`cmake ..`"
4. Execute "`make`"
5. Execute the simulation: "`./main`"

5.3 Detailed Description of the SystemC Braking Curves Model (URO)

Appendix

Fix a corrupted Papyrus model

Sometimes the Papyrus model can't be opened when a new Eclipse session is started. This can be the case if a model element containing a diagram was deleted. The problem is caused by a currupted di file. To repair the di file open it with an external text editor and remove all empty tags: <availablePage/> and <children/>. After that the model can be opened again in Papyrus.

[3] For more clarity we removed some debug outputs.