

UML and SystemC- a Comparison and Mapping Rules for Automatic Code Generation

Per Andersson
Lund University
P.O. Box 118
SE-221 00 Lund
Sweden
Per.Andersson@cs.lth.se

Martin Höst
Lund University
P.O. Box 118
SE-221 00 Lund
Sweden
Martin.Host@telecom.lth.se

Abstract

Today embedded system development is a complex task. To aid the engineers new methodologies and languages are emerging. During the development the system is modelled using different tools and languages. Transformations between the models are traditionally done manually. We investigate the automation of this process, specifically we are looking at automatic UML to SystemC transformation. In this paper we compare UML and SystemC, focusing on communication modelling. We also present mapping rules for automatic SystemC code generation from UML. The mapping has been implemented in our UML to SystemC code generator.

1 Introduction

Today there is a never ending demand for new functionality to be included in embedded systems such as mobile phones. This leads to increased design complexity. To overcome the increased system complexity new design methodologies, such as model driven architecture, have been introduced. In parallel with this, new languages, i.e. SystemC [2, 3], for system level modelling and simulation have also emerged. Combining new methodologies and new languages is a promising approach to manage the increasing system complexity. This is the focus of the MARTES (Model-Based Approach for Real-Time Embedded Systems development) project¹. In the project we investigate how UML and SystemC can be used together when the ideas of Model Driven Architecture are applied. One of the tasks of the project is to investigate how transformations from UML to SystemC can be automated and supported by tools. During this research we are developing a prototype tool, which manage

¹www.martes-itea.org

the UML to SystemC transformations and code generation, as an add-in to the Telelogic-TAU UML2 modelling tool². This part of the MARTES project is carried out in close cooperation between Lund University and Telelogic.

In this research there are a number of decisions that needs to be taken related to the detailed requirements on the developed tool. It is crucial to take the right decisions concerning what functionality to include in the tool. This is achieved by developing the tool iteratively. Different versions are developed after each other, and every version is evaluated in order to decide what additional functionality to include in the next version. Evaluations are a very important part of the development of the tool. The evaluations are being carried out together with other partners in the MARTES project, in the context of case studies in industrial projects.

In this paper we present the work and results from developing the first version of our UML to SystemC code generator. We start with a summary of related work in section 2. We compare the constructs and semantics of UML and SystemC in section 3. Based on this comparison we have developed a set of mapping rules which are detailed and motivated in section 4. Our implementation of the mapping rules is presented in section 5 and practical experience can be found in section 6. Finally the paper is concluded in section 7.

2 Related Work

Earlier publications on UML to SystemC mapping [4, 7] suggest that, to a large extent, there is a one to one relation between concepts in the two languages. For example a UML class is mapped to a SystemC module. This is not always desirable, sometimes UML classes are used for data encapsulation and in these cases they should remain as classes in

²www.telelogic.com

the SystemC model. Only UML classes with ports, and/or with architecture should be mapped to SystemC modules.

Riccobene et al. [7] address this by exposing all SystemC details in the UML model through a SystemC profile. Their approach is to use UML as an implementation language for SystemC. In addition to making all standard SystemC types available at the UML level they also extend actions in state machines to handle `sc_thread` and `sc_method` synchronisation. With their approach, the engineers must tag their UML model by adding relations to the intended SystemC elements. This is similar to the last part in our design process. In our design process engineers start with an abstract UML model, which is refined in three steps. This is further explained in section 5. One difference compared to their work is that we intend to automate most of this part in our process, minimising the design effort.

Another problem with bringing too many of the SystemC details into the UML model is the semantic differences between the languages, as discussed in section 3. This will lead to problems during co-simulation of pure UML models with models applying the SystemC profile. It will also be problematic to generate code for different targets, i.e. hardware and software. As far as we know no one has published a semantic comparison of these two languages.

3 Language Comparison

In this section we compare the UML and SystemC languages. The comparison is based on UML 2 [1, 6] and SystemC 2.2 [2, 3]. The purpose of the comparison is to find and motivate mapping rules for automatic SystemC code generation from UML. The focus is on concepts which are equivalent in both languages as well as concepts and constraints which are only present in one of the two languages.

When we refer to concepts which are similar in both languages we use the notation *UML_name/SystemC_name*, for example *class/module*. Also we refer to a class which inherits from `sc_module` as a SystemC module and any class inheriting from `sc_interface` as a SystemC interface.

3.1 Composition

UML and SystemC are similar from a structural point of view. Both languages have the concepts of *package/name space* which can be used to group most other language constructs. In real models packages/name spaces are mainly used to group declarations of *classes/modules*. A package/name space cannot be instantiated. Any instantiations done in a package/name space will result in one instance in the system, with limited visibility to the package/name space. In this paper we focus the discussion around the small system shown in figure 1 and 2. We will later show the SystemC code generated from it. In this system the package

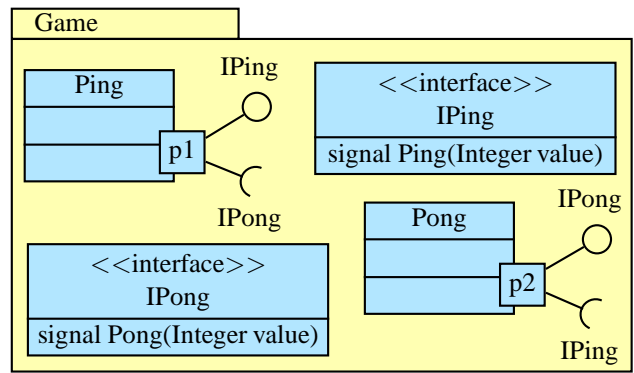


Figure 1. Structure in a UML design.

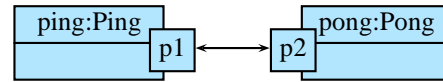


Figure 2. Communication in a UML design.

Game encapsulates the declarations of the classes `Ping` and `Pong` as well as the interfaces `IPing` and `IPong`.

A SystemC module is very similar to a UML class. In fact a SystemC module is defined as a C++ class with some predefined methods and attributes. Thus a SystemC module can have attributes and methods and also it can inherit from one or more classes and/or modules. There are a few properties which can be assigned to a UML class which cannot be expressed in the SystemC language. One such is *abstract*, but this can be emulated by making one of the methods in the class abstract. We believe the minor limitations of a SystemC module are negligible in practical use, so we treat UML classes as equivalent to SystemC modules.

Both UML classes and SystemC modules can contain references to other objects making it is possible to communicate between classes by method calls. This is however not the intended means to model communication in neither language. Instead communication should pass through *ports* connected using *connectors/channels*. A port is part of an class/module and defines its communication interface.

In UML a port has a required and a realised interface indicating which signals it will send and receive. Both the required and realised interface can be composed of a list of UML interfaces. Some tools also allow signal lists.

In SystemC a `sc_port` must have exactly one SystemC interface. The interface details which methods the module will call on the connected channel. A SystemC port corresponds to the required interface of a UML port. The equivalent of the UML realised interface is a SystemC `sc_export`. A SystemC `sc_export` is part of a module and has exactly one interface, which details the calls the module will implement.

3.2 Communication

The way communication is commonly modelled is quite different in UML and SystemC. In UML communication is modelled using *signals*, asynchronous messages that can carry data. The signals are sent through the ports of a class. The destination of a signal is determined by the *connectors* of the model. In figure 2 any signal sent from the object ping will be forwarded to the object pong. At the receiving object the signal is stored in a queue, from where it later will be consumed by the behaviour of this class. A UML connector only provides routing information for signals, it does not model the communication mechanism, i.e. a network or a bus. If this is to be included in the model it must be done using classes. Note also that a UML connectors are primarily a relation between two objects and not between classes. A UML port can have several connectors, and a connector connects exactly two ports.

SystemC channels are a central part of communication modelling in SystemC. They implement the behavior of the communication mechanism, as follows. During initialisation each port is connected to a channel. At this time the port saves a reference to the channel. Later during simulation the ports will be transparent, forwarding any operation to the channel (this is done by overriding the `->` operation). This design implies some constraints, a port may only connect to a channel which implement its interface, and also a port can only connect to one channel. By default there is no limit to how many ports that can connect to a channel, but it is possible for a channel to limit the number of ports connecting to it.

Since a message sending is realised as a method call in a channel, it is not possible for two modules to communicate without an intermediate channel. Also SystemC does not allow ports to be used to make methods in a module available to other modules. For this purpose `sc_export` was added to the language. Using `sc_export` a module can encapsulate a channel and export its interface to other modules. This makes it possible for a `sc_port` in one module to connect to a `sc_export` in another module without creating any intermediate channel.

4 Mapping Rules

Considering the difference in communication modelling it is clear that there does not exist a trivial, one to one mapping from UML to SystemC. Some UML constructs are however so similar to SystemC that we suggest that they should be replaced with the corresponding SystemC construct during the mapping process. Table 1 lists some of these constructs. We base our SystemC code generator to a large extent on Telelogic's C++ code generator [8]. This is possible since SystemC is a library and a simulation engine implemented

UML	SystemC
package	name space
active class	<code>sc_module</code>
class with ports	<code>sc_module</code>
other classes	C++ class

Table 1. Mapping rules for equivalent concepts.

on top of C++. The implementation of our code generator is explained further in section 5. In this section we focus on the mapping rules that are unique for SystemC and refer to [8] for details regarding mapping of the parts of the UML language not covered here.

The asynchronous communication of UML signals implies that there must exist a message queue somewhere in the communication. In UML this is located in the receiving class. When comparing to the predefined channels in SystemC, `sc_fifo` comes closest. However, there are some limitations which makes it less suitable. First, in a UML state machine it is possible to wait for one of several signals, i.e. several transitions, with different triggers, from the same state. When the state machine is in such state, the triggering signals might arrive on different ports. This leads to the need to do blocking reads on several fifo queues at the same time. This is not possible. In SystemC it is possible to wait for one of several events to occur, but when the call to wait returns it is not possible to determine which event that actually occurred. The concept of events in SystemC is similar to the `wait()` and `notify()` synchronisation mechanism found in, for example, Java. The second problem with `sc_fifo` is its limitation to connect only one sender and one receiver. This is the same constraint as a UML connector. The problem is that several UML connectors can connect to the same port, but a SystemC port can only connect to one channel. A UML connector does not provide any message queue, instead it connects the sending port with the queue inside the receiving class. This has the same semantic as connecting a SystemC port to a channel, assuming that the channel will provide a message queue.

The observation that a UML connector has the same semantic as connecting a SystemC port to a channel is one motivation for our mapping. We map a UML connector to code which connects the sending modules port with the channel containing the message queue of the receiving module. For this to work, we need a channel which implements a message queue and allows multiple connecting senders. Also, to solve the first problem with `sc_fifo`, this queue should be shared among all ports of the receiving module. There is no SystemC channel which meets these needs, so we generate one for each generated SystemC module. The

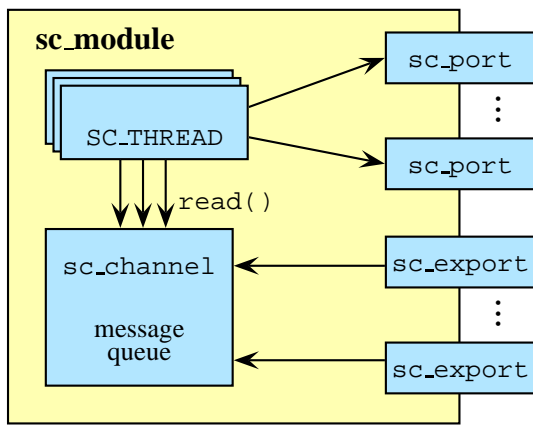


Figure 3. The structure of a generated SystemC module.

SystemC part	UML source
sc_port	port, required interface
sc_export	port, realised interface
sc_channel	interface, signal
sc_interface	interface, signal
constructor of sc_module	port, channel, state machine. Attribute initialisation have more sources
SC_THREAD	state machine

Table 2. The table lists SystemC parts and the UML constructs they are created from.

channel will handle all incoming signals to the module.

The structure of a generated SystemC module is shown in figure 3. The module is composed of one or more threads, one channel, and one or more ports and/or exports. When a message arrives at a **sc_export**, a method in the channel will be invoked and the message will be stored in the channels message queue. The threads in the module will later consume the message using the channels blocking `read()` method. The threads can also send messages through a **sc_port**. A message sent through a **sc_port** will arrive at a **sc_export** of another module.

Table 2 lists the UML sources for different SystemC constructs. How we generate the components of the SystemC module will be detailed below. For each realised interface of the UML class we generate one **sc_export** and connect it to the modules channel. The channel implements all realised interfaces with one method for each signal. The method stores the parameters of the signal in the channels message queue. This queue is implemented using

```

1 Ping ping("Ping");
2 Pong pong("Pong");
3 ping.p1_port(pong.p2_export);
4 pong.p2_port(ping.p1_export);

```

Figure 4. Code generated from figure 2.

a C++ `std::deque`. The channel also provides an blocking read, used by the modules threads, i.e. the methods generated from the state machine of the active UML class. To clarify, let us look at an example. The UML diagram in figure 2 will generate the SystemC code shown in figure 4. In the first two lines the module instances are created. Lines three and four connect the ports and exports of the generated module instances. Lines three and four are generated from the UML connector. Commonly line one and two will be attributes in a module and line three and four will be part of that modules constructor.

Next we will examine the SystemC declaration of module Ping, shown in figure 5. This originates from the UML view in figure 1. The UML port p1 is mapped to a **sc_port** and a **sc_export** at line 3-4. The interfaces IPing and IPong are generated from the UML interfaces. This mapping is detailed below. Lines 6-16 contain the declaration of the SystemC channel, which contains the message queue of the Ping module. The channel is instantiated at line 17. The method Ping at line 15 originates from the UML signal Ping and is part of the IPing interface inherited at line 8. The implementation is on lines 21-26. At line 30, in the constructor of Ping, `p1_export` is bound to the channel instance `Ping_channel`. With the generated code in figures 4 and 5, the module instance `pong` can send a `Ping` signal carrying the value three, using the syntax `p2_port->Ping(3)`.

4.1 Interfaces and Signals

The mapping of UML classes, ports and channels detailed above is not enough to generate code which compile. The SystemC interfaces and data structures for storing signals in the message queue are missing. These are generated from the UML interfaces and signal. For each UML interface a SystemC interface will be generated. The generated interface will contain one method for each signal in the UML interface. The method will have the same parameters as the original signal. In addition to the method each signal will also generate a class with one attribute for each signal parameter. The code generated from the UML interface IPing in figure 1 is shown in figure 7.

In SystemC a port and export can only have one interface and for a port to connect to an export/channel its interface must be implemented by the channel. Now look at figure 6.

```

1 SC_MODULE( Ping ){
2   public:
3   sc_export<IPing> pl_export;
4   sc_port<IPong> pl_port;
5   /*--- channel ---*/
6   class Channel_class:
7       public sc_channel,
8       public IPing{
9
10      private:
11      std::deque<UML_signal *> queue;
12      sc_event e;
13      public:
14      Channel_class(sc_module_name name);
15      UML_signal *read();
16      void Ping(int value);
17  };
18  Channel_class Ping_channel;
19  /*--- state machine behavior ---*/
20  void Ping_thread();
21  };
22  void Ping::Channel_class::Ping(int
23                                     value){
24      queue.push_back(new
25                      Ping_signal(value));
26      e.notify();
27  }
28  Ping(sc_module_name name) :
29      sc_module(name),
30      Ping_channel("Ping_channel"){
31      pl_export(Ping_channel);
32      SC_HAS_PROCESS(Ping);
33      SC_THREAD(Ping_thread);
34  }

```

Figure 5. Code generated from figure 1

UML port M1::p has three realised interfaces A, B, C. Ports M2::p and M3::p have required interfaces A, B and A, C. Port M1::p will be mapped to an SystemC export while M2::p and M3::p will generate SystemC ports. The generated export and ports need one interface each, here named if1, if2, and if3. A trivial attempt would be for the generated interfaces to inherit directly from A, B and C, i.e. `class if1:A, B, C{}`, `class if2:A, B{}`, and `class if3:A, C{}`. Now neither if2 nor if3 is a subtype of if1 and the ports can not connect to the export. A working solution is for if1 to inherit from if2 and if3. But now, due to the double inheritance, if1 will contain two instances of A. Also, with this mapping an exports interface will change as ports connect to it making it unpractical to generate code for parts of a system, or to distribute IP-cores in binary form, since they need to be recompiled when used.

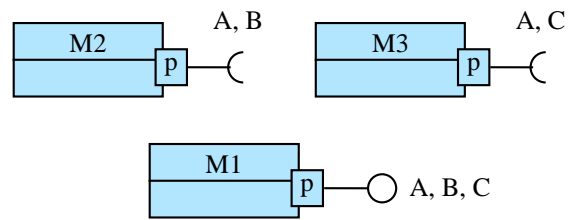


Figure 6. UML classes with interface lists.

```

1 class IPing: public sc_interface{
2   public:
3   virtual void Ping(int value) = 0;
4   class Ping_signal: public UML_signal{
5     public:
6     int value;
7     inline Ping_signal(int value) :
8         value(value){}
9   };
10 };

```

Figure 7. Code generated from figure 1

To solve this we suggest that one UML port should be mapped to several SystemC ports, one for each interface it requires. The list of realised interfaces can still be mapped to one export with one interface, i.e. `class if1:A, B, C{}`. With this mapping M2 will have two ports, one for interface A and one for B. Both can connect to M1::p. This solves the problems mentioned above while preserving the type hierarchy among interfaces in the UML model.

5 Mapping Process

During initial system modelling, a pure UML model is used. Though it is possible to define a set of mapping rules from a pure UML model directly into SystemC code, it would give the engineer little influence on the mapping and most likely a less satisfactory result. Instead we divide the mapping into three steps, as depicted in figure 8. All models are available and editable. This makes it possible for the engineer to have full control over the relevant details for the system under development and have the tool manage all remaining details.

Step 1, vertical refinement transformation: In this step an initial UML description is refined to a UML description, which follows a UML profile for SystemC. This step will, at least partly, be carried out manually. To minimise the design effort it should not be required to tag the whole model. This means that a set of default values for the SystemC specific attributes of the UML profile, must be defined. The default

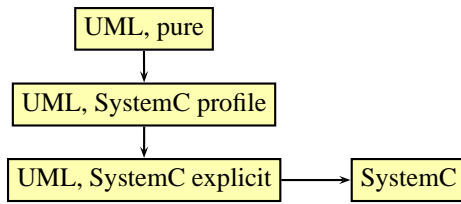


Figure 8. Our three step code generation.

values will in most cases provide a satisfactory mapping in the following transformation steps.

Step 2, vertical refinement transformation: In this step the model is transformed into a new UML description that only includes UML constructs with direct representations in SystemC, i.e. classes, attributes, inheritance etc. Other constructs such as state machines are translated to the target language. During this step we transform each state machine to a class with methods that implement the behaviour of the states and transitions. In the first version of the tool, the resulting model will be a un-timed functional model. The mapping rules for UML classes, ports, channels, signals and interfaces are given in section 4. A complete list of UML constructs which are removed during this transformation is beyond the scope of this paper.

In addition to removing UML only concepts, we also make all relations in the model explicit. When a class is made active in UML it implies that the class will have its own thread of execution. In SystemC this is realised using `SC_THREAD` or `SC_METHOD` which implies that the class is an instance of the SystemC class `sc_module`. During this transformation all such implicit relations are made explicit. For example, we add a generalisation relation to the SystemC class `sc_module` from all active UML classes. In the first version of the tool the resulting model is a un-timed functional model.

Step 3, horizontal transformation: In this step the UML model resulting from step 2 is transformed into a corresponding SystemC code. This transformation is a one to one correspondence between the UML model to the resulting SystemC code, i.e. this is a “pretty print” of the UML model. This step is implemented using the existing C++ code generator from Telelogic and thus reuse its support for scope rules, header-file inclusion and make file generation without any modifications. If the generated code is to be read by humans it is desirable to use the common SystemC macros when applicable. This requires a slight customisation of the syntax of the generated code. We do this using an *agent*, a mechanism which makes it possible for third party executables to interact with the C++ Code generator in Telelogic Tau G2. Our agent generates

SystemC like module declarations, instead of a C++ class declarations, `SC_MODULE(MyModule){...}` instead of `class MyModule:public sc_module{...}`.

6 Experimental Validation

The mapping rules and code generator presented in this paper have been use by VTT to extend their workload-based performance simulation [5]. The automatic mapping from UML to SystemC makes it possible to partially reuse existing UML application models, removing the need for separate work load models. VTTs experience is that our SystemC code generator is useful in practice and simplifies the engineers work in their model based design flow, see [5].

7 Conclusions

Combining new methodologies and new languages is a promising approach to overcome the increased complexity of today’s embedded systems. This is the driving force in the MARTES project. In this paper we compare UML and SystemC. The comparison reveals that the communication is modelled quite different in the two languages. Based on our observations we present mapping rules for automatic SystemC code generation from a UML model. We also present our transformation technique, composed of two vertical and one horizontal transformations. Using our transformation technique it is possible to reuse large parts of a code generator for other target languages similar to the target languages of the code generator, i.e the implementation of our SystemC code generator uses a large part of Telelogic’s C++ code generator.

References

- [1] H. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML 2 Toolkit*. OMG press, 2004.
- [2] T. Grötter, S. Liao, G. Marin, and S. Swan. *System Design With SystemC*. Kluwer Academic Publishers, 2002.
- [3] IEEE. IEEE Standard SystemC Language Reference Manual. *IEEE standard 1666-2005*, 2006.
- [4] D. Kathy, Z. S. Nguyen, and T. P.S. System driven SoC Design Via Executable UML to SystemC. *Real-Time Systems Symposium*, 2004.
- [5] J. Kreku, M. Hoppari, K. Tiensyrjä, and P. Andersson. SystemC workload model generation from UML for performance simulation. *Proceedings of Forum on specification and Design Languages (FDL)*, 2007.
- [6] D. Piltone and N. Pitman. *UML 2.0 In a Nutshell*. OMG press, 2004.
- [7] E. Riccobene, P. Scandurra, and S. Rosti, A. Bocchio. A SoC Design Methodology Involving a UML 2.0 Profile for SystemC. *Design Automation and Test Europe (DATE)*, 2005.
- [8] Telelogic, PO Box 4128, Kungsgatan 6, SE-203 12 Malmö, Sweden. *C++ Application Generator Reference*.