International Journal of Computer Applications (0975 – 8887)
Volume 8– No.6, October 2010

Automatic SystemC Code Generation from UML Models at Early Stages of Systems on Chip Design

Fateh Boutekkouk
Larbi ben M'hedi University
BP 358, Oum El Bouaghi, 04000
Algeria

ABSTRACT
In this paper, we present our approach for automatic SystemC code generation from UML models at early stages of Systems On Chip (SOC) design. A particularity of our proposed approach is the fact that SystemC code generation process is performed through two levels of abstraction. In the first level, we use UML hierarchic sequence diagrams to generate a SystemC code that targets algorithmic space exploration and simulation. In the second level of abstraction, messages that occur in sequence diagrams are implemented using UML activity diagrams whose actions are expressed in the C++ Action Language (AL) included in the Rhapsody environment from which a full SystemC code is generated for both simulation and synthesis.

General Terms
Systems On Chip, System Modeling

Keywords
UML, SystemC, Sequence diagrams, Activity diagrams, Action Language, Simulation.

1. INTRODUCTION
System On Chip (SOC) [1] can be defined as a complex integrated circuit that integrates the major functional elements of a complete end-product into a single chip or chipset.

The ever complexity of SOC design has pushed researchers in the field to raise the level of abstraction and exploit recent Software Engineering technologies such as object technology and in particular the Unified Modeling Language (UML) [2, 3].

SOC designers are now confronted with the challenge of how to bridge the gap between software standard modeling language such as UML and the well practiced SOC System Level Languages (SLL) like SystemC [4, 5, 6]. Since UML was initially introduced in the software domain, most commercial tools generate software code such as C, C++, and Java from UML models. However, there is a lack of tools that can synthesize UML models into SLL descriptions. Our objective is to raise the level from which SystemC descriptions can be generated to perform quick simulations and synthesis eventually. Thus a refinement directed approach seems inevitable to bridge the gap smoothly between UML models and SystemC descriptions. To address this problem, we have proposed a flow that permits automatic SystemC code generation from UML models at two levels of abstraction. The first level corresponds to SystemC code generation from UML sequence diagrams without implementing messages. Thus the generated code at this stage is oriented to algorithmic space exploration and simulation since the obtained code consists only of processes input/output ports, processes sensitivity lists, dependencies between processes, and signals. The second level of abstraction is a refinement of the first level where messages are implemented using UML activity diagrams whose state actions are expressed in the Action Language included in the Rhapsody environment [7]. At this stage, the generated code is dedicated to both simulation and synthesis. The rest of this paper is organized as follows: section two is dedicated to related works concerning the synthesis of UML models to SystemC code. Section three puts the light on UML. Section four gives an overview of the SystemC language. Our proposed flow with an illustrative example is discussed in section five. Section six is about implementation and a case study before concluding.

2. RELATED WORK
In this section, we try to present brievely some pertinent works targeting the generation of SystemC code from UML models.

SystemC code generation from UML was first investigated in [8], who presented several benefits when combining UML/SysML and SystemC, like a common and structured environment for system documentation/specification. The approaches in [9] and [10] cope with direct code generation by taking the UML model as an XMI (XML Metadata Interchange) file for translation to SystemC.
In [11], the authors presented a UML/SystemC profile for SystemC code generation from UML structural and Statecharts diagrams. In [12], the authors proposed a UML/MDA approach called *MoPCoM* methodology that permits automatic SystemC and VHDL code generation from UML models and MARTE profile by means of MDA techniques. Input models are focused on UML class, component, and Statecharts diagrams.
In [13], an approach to bridge the gap between UML and SystemC is presented. The proposed framework permits the integration of a customized SysML [16] (SysML) is an extension of UML for systems engineering entry with the code generation for HW/SW cosimulation and high level FPGA synthesis. Input models are focused on classes and blocks diagrams.
As opposite to these works, our approach tries to generate SystemC code automatically at early stages of SOC design (requirement analysis) from UML sequence diagrams in a first step then from UML activity diagrams in a second step.

3. THE UNIFIED MODELING LANGUAGE
UML [2] is a graphical object-oriented modeling language, initially, was used in software systems. However, and according to authors, UML can be tailored to SOC domain [14].

In our case, we have chosen the Rhapsody environment [7] for UML modeling, functional simulation, and automatic SystemC code generation.

4. SYSTEMC

SystemC [4] is an extension of C++ language for SOC modeling and simulation. It represents a standard for SOC system level modeling. Various versions of the language have appeared but we consider SystemC2.0.

SystemC structural designs are focused on modules.

A module contains ports, interfaces, channels, processes, and eventually other modules. In SystemC, concurrent behaviors are modeled using processes. A process has a sensitivity list that includes the set of signals to which it is sensitive. This list can be either static (pre-specified before simulation starts) or dynamic.

SystemC processes execute concurrently and may suspend on *wait()* statements. Such processes requiring their own independent execution stack are called "SC_THREADs". When the only signal triggering a process is the clock signal '*clk*' we obtain what we call "SC_CTHREAD" (clocked thread process). Certain processes do not actually require an independent execution stack and cannot suspended on *wait()* statement. Such processes are termed "SC_METHODs". SC_METHOD processes execute in zero simulation time and returns control back to the simulation kernel.

The following code [6] presents a SystemC module named *display* with an input port *din*, and a SC_METHOD called *print_data* which is sensible to *din*. For each SystemC module there are two files: *.h* for ports, functions, variables, and processes declaration and *.cc* for process and functions implementation. *systemc.h* designates the SystemC library file.

```
// display.h
#include "systemc.h"
#include "packet.h"
SC_MODULE(display) {
sc_in<long> din; // input port
void print_data();
// Constructor
SC_CTOR(display) {
SC_METHOD(print_data); // Method process to print data
sensitive << din;
}};
// display.cc
#include "display.h"
void display::print_data() {
cout <<"Display:Data Value Received, Data = "<< din <<
"\n";
```

5. OUR FLOW

As illustrated in figure 1, our proposed flow starts by capturing system requirements as a set of related uses cases and actors. At this stage, we use UML use cases diagrams with *'include'* and *'extend'* relations. Figure 2 gives an example of modelling with use cases diagram. In this example, we have one actor and two use cases named *usecase_0* and *usecase_1*. *usecase_0* is related to *usecase_1* by the 'include' relation. Each use case diagram is then refined to a set of interacting objects exhibiting a possible scenario. At this stage, we use UML sequence diagrams. The 'include' relation is modelled as an unconditional call of the use case child while the 'extend' relation is an optional call subject to some condition. Figure 3 shows a possible implementation of use cases using hierarchic sequence diagrams. In this example, we model *usecase_0* as the parent use case using sequence diagram with three interacting objects (class's instances) *class_0*, *class_1*, and *class_2* and an external object that represents the environment (*Env*). *usecase_1* is modelled as a child sequence diagram invoking by a call from the environment. In order to model the 'extend' relation, we add a conditional call invoking the child sequence diagram (*usecase_*2 in figure 4). From UML sequence diagrams, a SystemC code is generated automatically using the VB API which is integrated in the Rhapsody environment. This API offers the necessary functions and commands that permit the parsing of UML diagrams and then the extraction of information needed for SystemC code generation as text files. The generated code in this step will be used for algorithmic space exploration and simulation eventually. We have used three techniques for SystemC code generation process. In the first technique, each message is considered as a SystemC SC_METHOD. In the second technique, each end-to-end scenario is considered as a SystemC SC_THREAD. In the third technique, each object is considered as a SystemC SC_THREAD. Dashed lines in figure 1 enable the designer to modify his/her design according to simulation results.

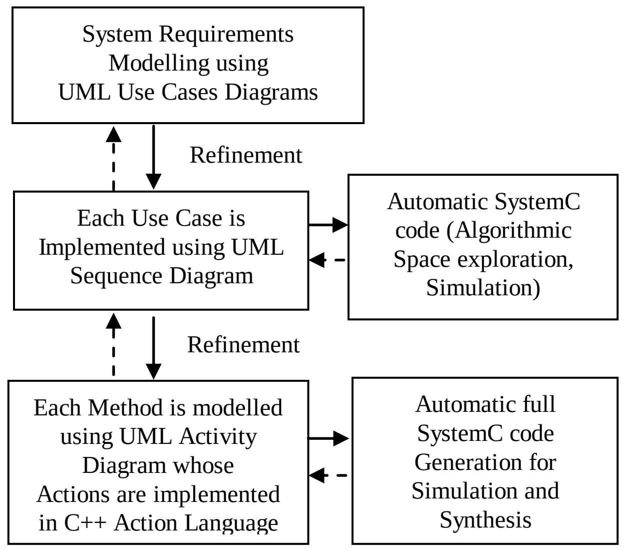

System Requirements Modelling using UML Use Cases Diagrams
Refinement
Each Use Case is Implemented using UML Sequence Diagram
Automatic SystemC code (Algorithmic Space exploration, Simulation)
Refinement
Each Method is modelled using UML Activity Diagram whose Actions are implemented in C++ Action Language
Automatic full SystemC code Generation for Simulation and Synthesis

Figure 1. Our proposed flow

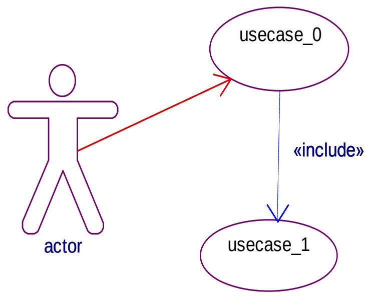


Figure 2. Example of UML use cases diagram

5.1 Illustrative example

In order to motivate our proposed approach, we apply the code generation process on an example whose use case diagram is illustrated in figure 2. In this example, we assume that we have an actor and two use cases named *usecase_0* and *usecase_1* that are related by an 'include' relation. Both *usecase_0* and *usecase_1* are implemented using UML sequence diagrams as showed in figure 5. In the following sections, we try to explain the three techniques for SystemC code generation from UML sequence diagrams.

5.2 First technique

In this technique, each message is mapped to a SystemC SC_METHOD. Methods arguments are transformed to input ports while returned values are mapped to output ports. To each call to a message, we add a Boolean input port that corresponds to the event to which process is sensible and a Boolean output port that corresponds to control return. From figure 5, we observe that *message_2* is used in both *usecase_0* and *usecase_1*. Such a common message will be mapped to a SC_METHOD process in a separate module. It is obvious, that this technique may lead to a very big number of fine grained processes which is not acceptable in complex designs. But it serves as a first solution for algorithmic space exploration. Table 1 shows the correspondence between UML and SystemC concepts.

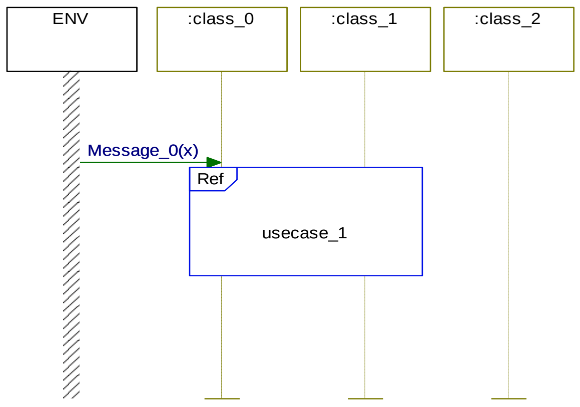


Figure 3. Possible implementation of 'include' relation

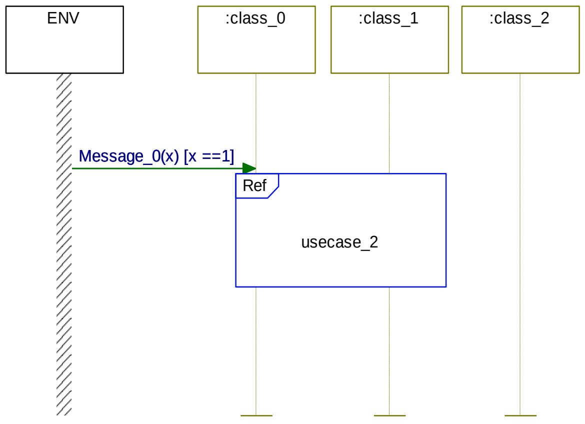
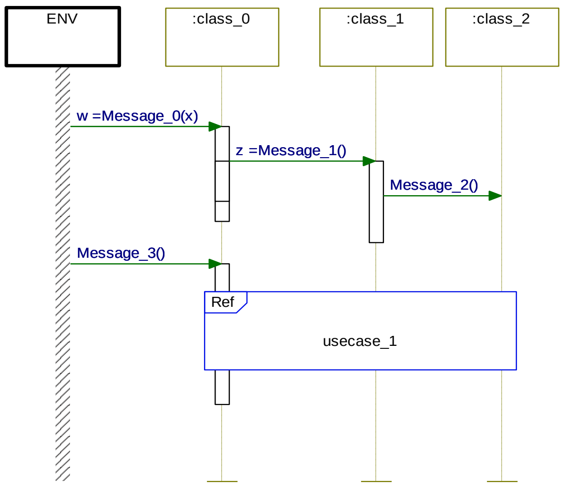
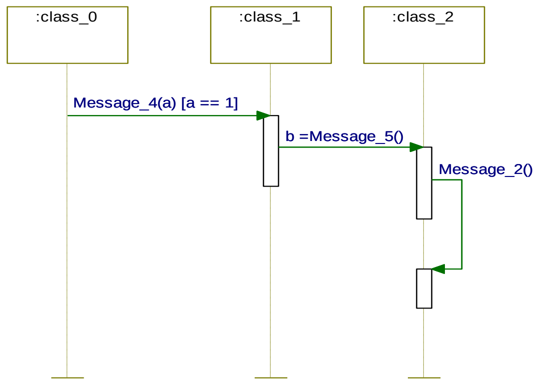


Figure 4. Possible implementation of 'extend' relation



(a)



(b)

Figure 5. Example of hierarchic sequence diagrams

(a) Parent sequence diagram (usecase_0); (b) Child sequence diagram (usecase_1)

Table 1. Correspondence between UML and SystemC for the first technique

UML concept	SystemC concept
message	SC_METHOD
Common message	SC_METHOD in a separate module
argument	sc_in<type> port
Return value	sc_out<type> port
call	sc_inout<bool>
Control return	sc_out<bool>
Child sequence diagram	Sub module
Top level model	*sc_main()*

Assume that we have a message with two integer arguments (*a* and *b*) and an integer return value *x*: *x = message(a, b).*
The corresponding SystemC code for this message is as follows:

```
// module1.h
# include "systemc.h"
SC_MODULE(module1){
sc_in<int> a;
sc_in<int> b;
sc_out<int> x;
sc_inout<bool> cal;
sc_out<bool> ret;
void message();
SC_CTOR(module1) {
SC_METHOD(message);
sensitive << cal; }};
// module1.cc
#include "module1.h"
void module1::message() {
int arg1, arg2, result;
while cal == 0 ;
cal = 0;      //  cal = false;
arg1 = a;
arg2 = b;
// message body
x = result;
ret = 1; }           //   ret = true;
```

SC_METHOD message is sensitive to the signal *cal*.

arg1 and *arg2* are two variables used to stock the two arguments coming from the two ports *a* and *b*.

result is a variable used to stock the returned value in the port *x*. We use the Boolean ports *cal* and *ret* to specify the message invoking and the return of the control to the caller respectively. The meaning of this SystemC code is as follows:

The process *message* will be blocked until the occurrence of the signal *cal* (*cal* = 1). After that, the process resumes its execution; sets *cal* to false; stocks the arguments coming from input ports *a* and *b* into variables *arg1* and *arg2*; performs some computation; stocks the result of computation into output port *x*; sets the signal *ret* to true. Similarly, The SystemC code for the caller is as follows:

```
// module2.h
# include "systemc.h"
SC_MODULE(module2){
sc_in<int> x;
sc_inout<bool> ret;
sc_out<int> a;
sc_out<int> b;
sc_out<bool> cal;
void caller();
SC_CTOR(module2) {
SC_METHOD(caller);
sensitive << ****; // some ports
}};
// module2.cc
#include "module2.h"
void module2::caller() {
int result;
// instructions;
cal = 1;      //  cal = true;
a = " ";  // arguments initialization
b = " ";
While ret == 0 ;
ret = 0;
result = x;
// remaining instructions
}
```

Note that SC_METHOD processes *message* and *caller* are put in two distinct modules: *module1* and *module2* respectively. However, if we put them into one module, all ports become sc*_inout*. By applying this technique on our example, we obtain six (6) SC_METHOD processes that are: *Message_0*, *Message_1*, *Message_2*, *Message_3*, *Message_4*, and *Message_5*. Assume that all messages arguments and return values are integers. *cal0*, *cal1*, *cal2*, *cal3*, *cal4*, and *cal5* designate Boolean ports for *message_0*, *message_1*, *message_2*, *message_3 message_4*, and *message_5* calls respectively. *arg0* and *arg4* designate ports for *message_0* and *message_4* arguments respectively. *val0*, *val1*, and *val5* designate ports for *message_0*, *message_1*, and *message_5* returned values respectively. *ret0*, *ret1*, *ret2*, *ret3*, *ret4*, and *ret5* designate Boolean ports for messages controls return.

Since *message_2* is a common message, we put it in a separate module called *mess2*. Here, we have two modules: *usecase0* including SC_METHODS *message_0*, *message_*1, and *message_3*, and *usecase1*including *message_4*, and *message_5.*

The corresponding SystemC code looks like:

```
// mess2.h
# include "systemc.h"
```

```
SC_MODULE(mess2){
sc_inout<bool> cal2;
sc_out<bool> ret2;
void message_2();
SC_CTOR(mess2) {
SC_METHOD(message_2);
sensitive << cal2;
}};
// mess2.cc
#include "mess2.h"
void mess2::message_2() {
while cal2 == 0 ;
cal2 = 0;
// message body;
ret2 = 1;}

// usecase1.h
# include "systemc.h"
SC_MODULE(usecase1){
sc_in<int> arg4; sc_inout<int> val5; sc_out<bool> cal2;
sc_inout<bool> ret2; sc_inout<bool> cal4;
sc_inout<bool> cal5; sc_inout<bool> ret5; sc_out<bool> ret4;
void message_4(); void message_5();
SC_METHOD(message_4);
sensitive << cal4;
SC_METHOD(message_5);
sensitive << cal5;
}};

// usecase1.cc
void usecase1::message_4() {
int var, result;
while cal4 == 0;
cal4 = 0;
var = arg4;
// instructions
cal5 = 1;
while ret5 == 0;
ret5 = 0;
result = val5;
// remaining instructions
ret4 = 1;
}
void usecase1::message_5() {
// code
}
// usecase0.h
# include "systemc.h"
SC_MODULE(usecase0){
sc_in<int> arg0; sc_inout<int> arg4; sc_out<int> val0;
sc_inout<int>   val1;   sc_inout<bool>   cal0;   sc_inout<bool>
cal1;
sc_out<bool> cal2; sc_inout<bool> cal3; sc_out<bool> cal4;
sc_out<bool> ret0; sc_inout<bool> ret1; sc_inout<bool> ret2;
sc_out<bool> ret3; sc_inout<bool> ret4;
void message_0(); void message_1(); void message_3();
SC_CTOR(usecase0) {
SC_METHOD(message_0);
sensitive << cal0;
SC_METHOD(message_1);
sensitive << cal1;
SC_METHOD(message_3);
sensitive << cal3;
}};

// usecase0.cc
#include "usecase0.h"
void usecase0::message_0() {
 // code
};
void usecase1::message_1() {
// code
};
void usecase1::message_3() {
int var;
while cal3 == 0 ;
cal3 = 0;
// instructions
arg4 = var;
if arg4 = 1 {
cal4 = 1;
while ret4 == 0;
ret4 = 0;
}
// remaining instructions
ret3 = 1;
};
// main.cc
#include "mess2.h"
#include "usecase1.h"
#include "usecase0.h"
int  sc_main(int argc, char* argv[]) {
sc_signal<int> ARG0, ARG4, VAL0, VAL1;
sc_signal<bool> CAL0, CAL1, CAL2, CAL3, CAL4, CAL5 ;
sc_signal<bool> RET0, RET1, RET2, RET3, RET4, RET5 ;
mess2 ms2("mess2"); ms2.cal2(CAL2);ms2.ret2(RET2);
usecase1 uc1("usecase1");
uc1.arg4(ARG4);uc1.val5(VAL5);uc1.cal2(CAL2);
uc1.cal4(CAL4);uc1.cal5(CAL5);uc1.ret2(RET2);uc1.ret4(RET4)
;uc1.ret5(RET5);
usecase0 uc0("usecase0");
uc0.arg0(ARG0);uc0.arg4(ARG4);uc0.val0(VAL0);
uc0.val1(VAL1);uc0.cal0(CAL0);uc0.cal1(CAL1);
uc0.cal2(CAL2);uc0.cal3(CAL3);uc0.cal4(CAL4);
uc0.ret0(RET0);uc0.ret1(RET1);uc0.ret2(RET2);
uc0.ret3(RET3);uc0.ret4(RET4);
return(0);}
```

5.3 Second technique

In this technique, we consider each end-to-end scenario as a SystemC SC_THREAD. An end-to-end scenario is a sequence of methods that are invoked by an external call from the environment. Table 2 shows the correspondence between UML and SystemC concepts. All internal methods are implemented as SystemC functions.

Table 2. Correspondence between UML and SystemC for the second technique

UML concept	SystemC concept
End-to-end scenario	SC_THREAD
Internal message	C++ function
External call	port
Top level model	*sc_main()*

By applying this technique on the above example, we obtain two SystemC SC_THREADS: *process1* including the sequence of messages: *message_0*, *message_1*, and *message_2* and *process2* including *message_3*, *message_4*, *message_5*, and *message_2*. *process1* is sensitive to *cal0* and *process2* to *cal3*.

The corresponding SystemC code is as follows:

```
// system.h
# include "systemc.h"
SC_MODULE(system){
sc_in<int> arg0; sc_inout<bool> cal0; sc_inout<bool> cal3;
sc_out<bool> ret0; sc_out<bool> ret3; sc_out<bool> val0;
int message_0(int);  int message_1(void) ; void message_2(void);
void message_3(void); void message_4(int); int
message_5(void);
void process1(); void process2();
SC_CTOR(system) {
SC_THREAD(process1);
sensitive << cal0;
SC_THREAD(process2);
sensitive << cal3;
}};
// system.cc
void message_2(void){
// message_2 body}

int message_1(void){
// instructions
message_2() ;  // call to message_2
// remainig instructions}
int message_0(int) {
int result;
// instructions
Result = message_1();
// remaining instructions
return}
int message_5(void) {
// instructions
message_2() ;
// remaining instructions
Return}

void message_4(int) {
int result ;
// instructions
Result = message_5() ;
// remaining instructions}
void message_3(void) {
int arg ;
// instructions
if arg == 1 message_4(arg) ;
// remaining instructions}

void system::process1() {
wait();
cal0 = 0;
arg = arg0;
val0 = message_0(arg);
ret0 = 1; }
void system::process2() {
wait();
cal3 = 0;
message_3();
ret3 = 1; }

// main.cc
#include "system.h"
int  sc_main(int argc, char* argv[]) {
sc_signal<bool> CAL0, CAL3, RET0, RET3;
sc_signal<int> ARG0,VAL0;
system  sys("system");
sys.arg0(ARG0);sys.cal0(CAL0);sys.cal3(CAL3);
sys.ret0(RET0);sys.ret3(RET3); sys.val0(VAL0);
return(0); }
```

5.4 Third technique

In this technique, each UML object is considered as a SC_THREAD. For each input /output message call, we create input/output ports (we add more ports for arguments and returned values). Table 3 shows the correspondence between UML and SystemC concepts. By applying this technique on the above example, we obtain four processes (4): *Env*, *class_0*, *class_1*, and *class_2*.

Table 3. Correspondence between UML and SystemC for the third technique

UML concept	SystemC concept
Object	SC_THREAD
Input message call	Input ports

output message call	Output ports
Top level model	*sc_main()*

For the sake of space, we give only the SystemC code for *Env* and *class_0*.

```
// system.h
# include "systemc.h"
SC_MODULE(system){
sc_inout<bool> cal0 ; sc_inout<bool> cal1;
sc_inout<bool> cal2; sc_inout<bool> cal3;
sc_inout<bool> cal4; sc_inout<bool> cal5;
sc_inout<bool> ret0; sc_inout<bool> ret1;
sc_inout<bool> ret2; sc_inout<bool> ret3;
sc_inout<bool> ret4; sc_inout<bool> ret5;
sc_inout<int> arg0, arg4,val0, val1, val5;
void env();
void class_0(); void class_1(); void class_2();
SC_CTOR(system) {
SC_THREAD(env);
sensitive << ret0 << ret3 ;
SC_THREAD(class_0);
sensitive << cal0 << ret1 << cal3 << ret4 ;
SC_THREAD(class_1);
sensitive << cal1 << ret2 << cal4 << ret5 ;
SC_THREAD(class_2);
sensitive << cal5 << cal2 ;}};
// system.cc
#include "system.h"
void system::env() {
int temp;
cal0 = 1;
arg0 = 1; // some initialization
wait (ret0);
ret0 = 0;
temp = val0;
cal3 = 1;
wait (ret3);
ret3 = 0;
}
void system::class_0() {
int arg, temp;
wait (cal0);
cal0 = 0;
arg = arg0;
-- message0 instructions
cal1 = 1;
wait (ret1);
ret1 = 0;
-- remaining message_0 instructions
ret0 = 1;
val0 = w;
wait (cal3);
cal3 = 0;
-- message3 instructions
temp := a;
if temp = 1{
cal4 = 1;
wait (ret4);
ret4 = 0;}
-- remaining message_3 instructions
ret3 = 1;
}
void system::class_1() {
// body of class_1
}
void system::class_2() {
// body of class_2
}
// main.cc
#include "system.h"
int  sc_main(int argc, char* argv[]) {
sc_signal<bool> CAL0, CAL1, CAL2, CAL3, CAL4, CAL5;
sc_signal<bool> RET0, RET1, RET2, RET3, RET4, RET5;
sc_signal<int> ARG0,ARG4,VAL0,VAL1, VAL5;
system sys("system");
sys.arg0(ARG0);sys.arg4(ARG4);sys.val0(VAL0);
sys.val1(VAL1);sys.val5(VAL5);                    sys.cal0(CAL0);
sys.cal1(CAL1);                  sys.cal2(CAL2);sys.cal3(CAL3);
sys.cal4(CAL4);           sys.cal5(CAL5);          sys.ret0(RET0);
sys.ret1(RET1); sys.ret2(RET2); sys.ret3(RET3); sys.ret4(RET4);
sys.ret5(RET5);
return(0) ;}
```

5.5 Modeling with UML activity diagrams

In our proposed flow (see figure 1), the second step consists in internal behaviour modelling of messages using UML activity diagrams whose state actions are expressed in the C++ Action Language (AL) included in the Rhapsody environment. The AL is a subset of C++ that uses a C++ compiler to enable the model simulation. This language provides message passing, data checking, actions on transitions, and model execution. It supports majority of C++ operators, *if*/*else*, *for*, *while*, *do*/*while*, *return* instructions, primitive types, array of primitives, objects, invoking block operations, generating events, generating port events, testing port for an event, etc. Using the Rhapsody environment, we can perform functional simulation before code generation. This step is very important to validate the SystemC code functionality against UML functional models.

6. IMPLEMENTATION AND CASE STUDY

We have used the Rhapsody environment for UML modelling and SystemC code generation. In order to automate the SystemC code generation from UML models, we have used the VB API which is integrated in the Rhapsody environment. With VB, we can easily parse UML graphical models then collect the necessary information to create SystemC files. We have developed a VB macro for SystemC code generation and integrated it as a tool box in the Rhapsody environment. As a case study, we have chosen the SDP (Simplex Data Protocol) [6] application whose UML main sequence diagram is illustrated in figure 6. Figure 7 gives us an overview of generated SystemC files for the *receiver* object.

7. CONCLUSION AND PERSPECTIVES

In this paper, we present our approach for automatic SystemC code generation from UML models at early stages of SOC design. Our proposed flow consists mainly of two steps: generation of SystemC codes from UML hierarchic sequence diagrams then from UML activity diagrams. Actions of activity diagrams are expressed in the C++ Action Language (AL) which is included in the Rhapsody environment. From AL, a full SystemC code is generated for both simulation and synthesis. SystemC code is generated as text files automatically and this is due to the VB API included in the Rhapsody environment. As a perspective, we plan to investigate the MDA approach for SystemC code generation from Sequence diagrams and consider asynchronous events and temporal constraints.

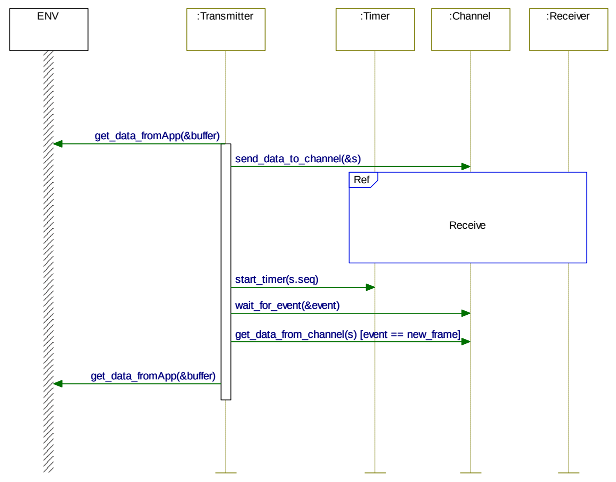
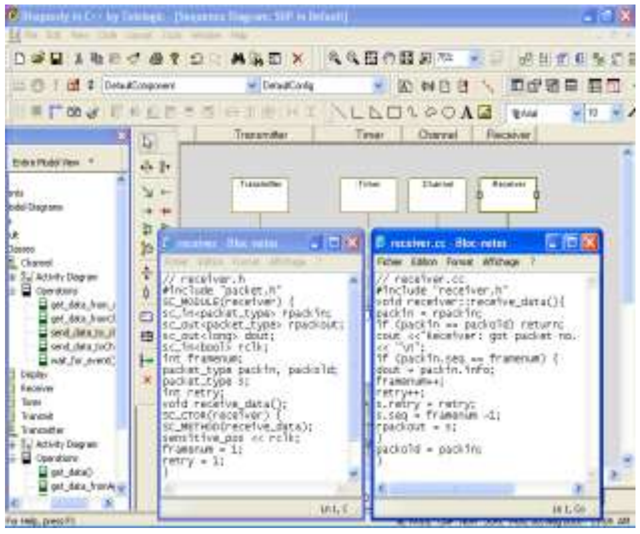


Figure 6. UML sequence diagrams for SDP



```
// receiver.h
#include "packet.h"
SC_MODULE(receiver) {
sc_in<packet_type> rpackin;
sc_out<packet_type> rpackout;
sc_out<long> dout;
sc_in<bool> rclk;
int framenum;
packet_type packin, packold;
packet_type s;
int retry;
void receive_data();
SC_CTOR(receiver) {
SC_METHOD(receive_data);
sensitive_pos << rclk;
framenum = 1;
retry = 1;
}
```

```
// receiver.cc
#include "receiver.h"
void receiver::receive_data(){
packin = rpackin;
if (packin == packold) return;
cout <<"Receiver: got packet no.
<< "\n";
if (packin.seq == framenum) {
dout = packin.info;
framenum++;
retry++;
s.retry = retry;
s.seq = framenum -1;
rpackout = s;
}
packold = packin;
}
```

Figure 7. SystemC code generation from Rhapsody UML models

8. REFERENCE

[1] Jerraya, A.A. and Wolf, W. 2005. Multiprocessor systems on chip. Morgan Kaufmann publishers.

[2] Booch, G., Rumbaugh, J. and Jacobson I. 1999. Unified Modeling Language User Guide (Addison-Wesley).

[3] UML2.0 Superstructure Specification. 2003, http://www.omg.org

[4] SystemC, IEEE Standard SystemC® language Reference Manual. 2005, www.systemc.org

[5] SystemC, Functional specification for SystemC 2.0. 2002, www.systemc.org

[6] SystemC, Version 2.0 User’s guide, 2002, www.systemc.org

[7] Rhapsody UML modeler, www.telelogic.com/products/rhapsody, from Telelogic, an IBM company.

[8] Pauwels, M., et al. A design methodology for the development of a complex system-on-chip using UML and executable system models. 2004. In System Specification & Design Languages. Springer US.

[9] Nguyen, K. D. Sun, Z., Thiagarajan, P. and Wong W.-F. 2004. Model-driven SOC design via executable UML to SystemC. In IEEE RTSS’04.

[10] Tan, W., et al. Synthesizable SystemC code from UML models. 2004. In UML for Soc Design, DAC 2004 Workshop.

[11] Riccobene, E., Scandura, P., Rosti, A. and Bocchino, S. 2005. A SOC Design Methodology Involving a UML2.0 Profile for SystemC. Proceedings of the Design, Automation and Test in Europe Conference end Exhibition (DATE’05).

[12] Vidal, J., De Lamotte, F., Gogniat, G., Soulard, P. and Diguet, JP. 2009. A codesign approach for embedded system modeling and code generation with UML and MARTE. In DATE09.

[13] Mischkalla, F., He, Da., and Mueller, W. Closing the Gap between UML-based Modeling, Simulation and Synthesis of Combined HW/SW Systems. 2010. In DATE10.

[14] Boutekkouk, F., Benmohammed, M., Bilavarn, S. and Auguin, M. 2009. UML2.0 profiles for Embedded Systems and Systems On a Chip (SOCs). In JOT (Journal of Object Technology).

[15] Coyle, F.P, Thornton, M.A. 2005. From UML to HDL: a Model Driven Architectural Approach to Hardware-Software Co-Design. Proceedings of Information Systems: New Generations Conference (ISNG), p. 88-93.

[16] Systems Modeling Language (SysML) Specification. 2006. OMG document: ad/2006-03-08-01, version 1. Draft.