# Bridging the Gap between UML and Hardware Description Languages at Early Stages of Embedded Systems Development

Fateh Boutekkouk[1]

[1]Department of computer science, Larbi Ben M'hedi University,
BP 358, Route de Constantine, Oum El Bouaghi, 04000, Algeria
fateh_boutekkouk@yahoo.fr

***Abstract***: This work deals with automatic Hardware Description Languages (HDLs) code generation from UML 2.0 models at early stages of embedded systems development. In our case, we target two standard HDLs which are SystemC and VHDL. A particularity of our proposed approach is the fact that HDLs code generation process is performed through two levels of abstraction. In the first level, we use UML hierarchic sequence diagrams to generate a HDL code that targets algorithmic space exploration and simulation eventually. In the second level of abstraction, messages that occur in sequence diagrams are implemented using UML activity diagrams whose state actions are expressed in the C++ Action Language included in the Rhapsody environment from which a full HDL code is generated for both simulation and synthesis. We have developed two macros for SystemC and VHDL code generation and integrated them as tool boxes in the Rhapsdoy environment.

***Keywords***: Embedded Systems, UML, SystemC, VHDL, Simulation, Synthesis**.**

## 1. Introduction

We can define Embedded Systems (ESs) [9] as application-specific computers, masquerading as non-computers that interact with the physical world and must perform a small set of tasks cheaply and efficiently. ESs have specific characteristics such as heterogeneity (hardware / software), ability to react, criticality, real time, and consumption constraints.

Modern ESs are able to execute very complex algorithms ranging from control, telecommunication to media high performance applications implemented in only one chip (SOC: System-On-a-Chip) [10].

The ever complexity of embedded systems (ESs) 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) [6].

ESs designers are now confronted with the challenge of how to close the gap between UML and the well practiced Hardware Description Language (HDL) in ESs world such as SystemC [20] and VHDL [23].

Since UML was originally introduced in the software field, 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 HDL descriptions.

Our objective is to raise the level from which HDL descriptions can be generated to perform quick algorithmic space exploration, simulation and synthesis eventually. Thus a refinement directed approach seems inevitable to bridge the gap smoothly between UML models and HDLs descriptions.

To address this problem, we have proposed a flow that permits automatic HDL code generation from UML models at two levels of abstraction. The first level corresponds to HDL code generation from UML sequence diagrams without implementing messages. Thus the code generated at this stage is oriented to algorithmic space exploration and simulation eventually 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 viewed as a refinement of the first level where messages are implemented using UML activity diagrams whose actions are expressed in the C++ Action Language included in the Rhapsody environment [15]. At this stage, the generated code is dedicated to both simulation and synthesis. In this paper, our main contribution is the development of a tool that can generate SystemC and VHDL code from UML models following a refinement directed approach. The rest of this paper is organized as follows: section two is dedicated to related works concerning the synthesis of UML models to SystemC and VHDL code. Section three gives an overview of VHDL and SystemC languages. Our proposed flow with an illustrative example is discussed in section four. The implementation of our tool and a case study is discussed in section five before concluding.

## 2. Related Work

In this section, we try to present briefly some pertinent woks targeting the generation of VHDL and SystemC codes from UML models.

The authors in [9] proposed the synthesis of state diagrams into VHDL.

In [12], the authors presented a technique for generating VHDL descriptions from a subset of UML, and a set of rules to transform UML classes and Statecharts to VHDL.

The authors in [4] and [5] used SMDL (the language with formal semantics and high-level concepts such as states, queues and events) as an intermediary language to generate VHDL code from UML Statecharts and activity diagrams.

A Model Driven Architecture (MDA) approach for generating VHDL code from UML models was proposed in [1], [8], and [17]. In [8], the authors used UML Meta-model to generate different platform specific implementations.
In [17], the authors defined a set of rules to map UML to VHDL in a practical code generator.
In [16], the authors presented a UML/SystemC profile for SystemC code generation from UML structural and Statecharts diagrams.
In [21], the authors developed a tool for UML synthesis called: *Chip Fryer* that can generate VHDL code from XMI representation of UML models. The input model consists of class, object diagrams, and state machines. Actions are expressed in a C++ action language.
In [24], the authors proposed a UML/MDA approach called *MoPCoM* methodology that permits automatic VHDL and SystemC code generation from UML models and MARTE profile by means of MDA techniques. Input models are focused on UML class, component, and Statecharts diagrams.
Contrary to these works, our approach tries to generate VHDL and SystemC codes automatically at early stages of ESs development from UML sequence diagrams in a first step then from UML activity diagrams in a second step.

## 3. VHDL and SystemC

### 3.1 VHDL

VHDL (VHSIC Hardware Description Language) [2], [3], [23] is an industrial standard HDL. It looks similar to programming language ADA and used for both simulation and synthesis.
Now VHDL is governed by IEEE standards and very popular for European design houses. VHDL models consist of an external part (entity) that defines the Inputs/Outputs of the model and the internal part that describes the operation of the model (the architecture). The Entity declaration format looks like:

*entity entity_name is*
*port (signal_name(s): mode signal_type;*
*:*
*signal_name(s): mode signal_type);*
*end entity entity_name;*

*mode* describes the direction of transferred data through port (*in*, *out*, or *inout*); *signal_type* defines the signal(s) type.
The Architecture format looks like:

*architecture architecture_name of entity_name is*
*begin*
*:*
*end architecture architecture_name;*

VHDL designs can be written in three different styles: structural, data flow, and behavioural. Of course, these three styles can be mixed. Structural descriptions describe the interconnection of hierarchy and are useful for designs reuse. They consist of component instantiation statements (i.e. *port map* instruction) which are concurrent statements.
Behavioural descriptions are focused on the *process* concept. The latter is used in two ways:
For combinational logic, we mention the list of all process input signals after the keyword *process*. The general form is:

*process (signal_names)*
*begin*
.....
*end process;*

For sequential logic, two cases occur:
In the first case, the sensitivity list is empty, but statements inside the process must include wait statements;
In the second case, the sensitivity list contains the clock signal and the statements are within an *if* statement.
The general form is as follows:

*process (clock)*
*begin*
*if clock and clock'event then*
....
*end if;*
*end process;*

Processes communicate via signals. Many processes can be put in one architecture. VHDL supports classical language data types such as: *boolean*, *character*, *integer*, *real*, and *string* and control statements such as *if*, *loop*, and *case*. In addition, VHDL has the types: *bit*, *bit_vector*, and the IEEE 1164-standard-logic types that are *std_logic* and *std_logic_vector*. For more details on VHDL, one can refer to [23].

### 3.2 SystemC

SystemC [18], [19], [20] is an extension of C++ language for SOC modeling and simulation. 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 [19] presents a SystemC module named *display* with an input port *din*, and an 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";
```

## 4. Our proposed flow

As showed 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 uses cases 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 into a set of interacting objects showing a possible scenario. At this stage, we use UML sequence diagram. 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, VHDL and SystemC codes are generated automatically using the VB API which is integrated in the Rhapsody environment. This API offers the necessary functions and commands that permit the manipulation of UML diagrams and then the extraction of information needed for HDL 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 HDL code generation process:
1st technique: each message is considered as a VHDL process/SystemC SC_METHOD.
2nd technique: each end-to-end scenario is considered as a VHDL process/ SystemC SC_THREAD.
3rd technique: each object is considered as a VHDL process/ SystemC SC_THREAD.
For each technique, two styles of VHDL descriptions are generated: structural using VHDL mapping instructions and behavioural using the VHDL process concept. Dashed lines in figure 2 enable the designer to modify his/her design according to simulation results. VHDL/SystemC simulation and/or synthesis are performed using available commercial tools such as *ModelSim* (ModelSim) or *SystemC* simulator.

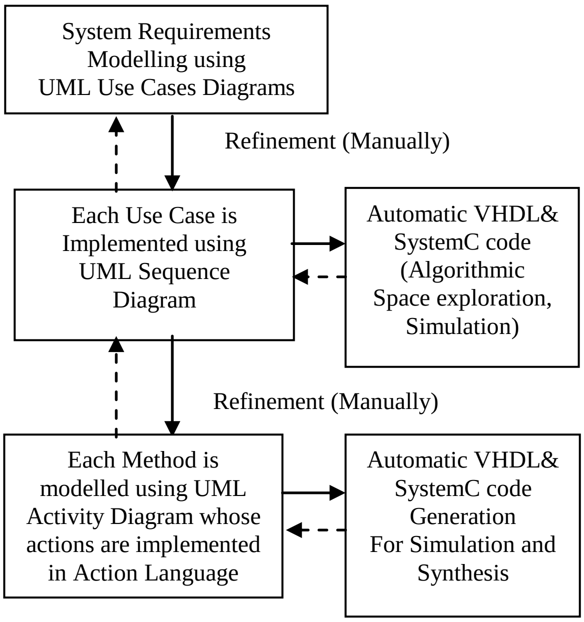



**Figure 1.** Our proposed flow

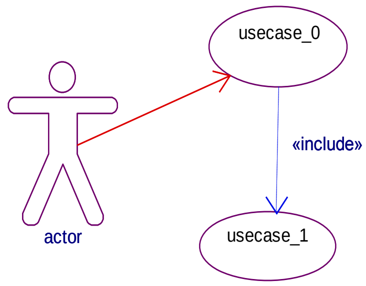



**Figure 2.** Example of UML use cases diagram

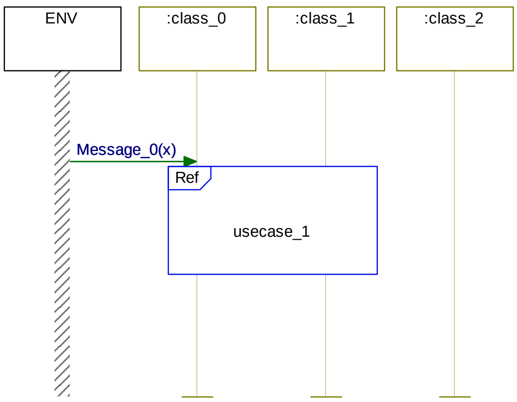



**Figure 3.** Possible implementation of 'include' relation

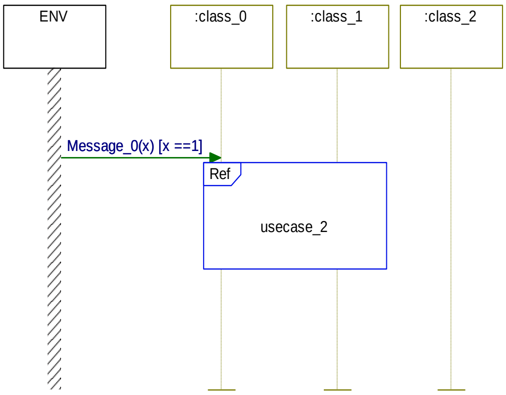


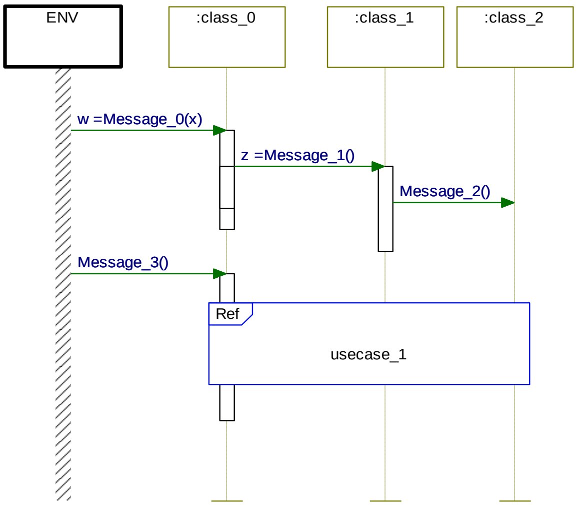

**Figure 4.** Possible implementation of 'extend' relation

### 4.1 Illustrative example

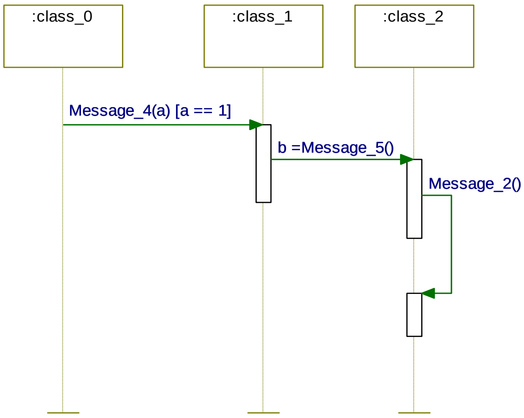
In order to motivate our proposed approach, we try to apply the HDL code generation process on an example whose use case diagram is illustrated in figure 2. In this example, we assume that we have an actor and two use cases named *usecase_0* and *usecase_1* that are related by an 'include' relation. 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 VHDL/SystemC code generation from UML sequence diagrams.

### 4.2 First technique

In this technique, each message is mapped to a VHDL process or 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. Two styles of VHDL descriptions are generated: the behavioural description and structural description. In the former, all generated processes from children sequence diagrams are put in one architecture that corresponds to the main sequence diagram. In the latter, we consider children sequence diagrams as sub entities reflecting the hierarchy of the design. Table 1 shows the correspondence between UML and VHDL/SystemC concepts.




(a)




(b)

**Figure 5.** Example of hierarchic sequence diagrams (a) parent sequence diagram (usecase_0); (b) child sequence diagram (usecase_1)

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 VHDL code for this message is as follows:

```
message : process is
variable arg1, arg2, result : integer;
begin
wait until cal = true;   -- cal read
cal <= false;            -- cal write
arg1 := a;
arg2 := b;
-- message body
x <= result;             -- x write
ret <= true;             -- ret write
end process message;
```

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

| UML concept | VHDL (behavioral /structural) | SystemC |
|---|---|---|
| Message | Process/Entity | SC_METHOD |
| Common message | Process/Entity | SC_METHOD in a separate module |
| Argument | in port | sc_in <type> port |
| Returned value | out port | sc_out <type> port |
| call | inout port (boolean) | sc_inout <bool> port |
| Control return | out port (boolean) | sc_out <bool> port |
| Child sequence diagram | sub entity (structural) | sub module |
| Top level model | Test bench | sc_main() |

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

```
caller : process is
variable val : integer;
begin
-- instructions
cal <= true;              -- cal write
a <= " ";                 -- initialization
b <= " ";
wait until ret = true;    -- ret read
ret <= false;             -- ret write
val:= x;                  -- x read
-- Remaining instructions
end process caller;
```

The meaning of this VHDL code is as follows:
After performing some computations, the process *caller* sets the signal *cal* to true; initializes the arguments ports *a* and *b*; blocked until the occurrence of the signal *ret* (*ret* = true). After that, the process resumes its execution: sets *ret* to false; stocks the content of port *x* into variable *val*; performs the remaining computation.
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 var1, var2, result;
while cal == 0 ;
cal = 0;     // cal = false;
var1 = a;
var2 = b;
// message body
x = result;
ret = 1; }          //  ret = true;
```

SC_METHOD message is sensitive to the signal *cal*.
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) VHDL processes and six SC_METHOD processes that are: *Message_0*, *Message_1*, *Message_2*, *Message_3*, *Message_4*, and *Message_5*. In the VHDL behavioural style, all processes are put in one architecture. The entity includes all processes ports. 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.
The corresponding VHDL code for the behavioural description is as follows:

```
entity  usecase_0 is
port (cal0, cal1, cal2, cal3, cal4, cal5 : inout boolean; arg0 : in integer; arg4 : inout integer; ret0, ret1, ret2, ret3, ret4, ret5 : inout  boolean; val0: out integer; val1, val5 : inout integer);
end entity usecase_0;

architecture system of usecase_0 is
begin
message_0 : process is
variable arg, val : integer;
begin
wait until cal0 = true;
cal0 <= false;
arg := arg0;
-- instructions
cal1 <= true ;
wait until ret1 = true ;
ret1 <= false ;
val := val1;
-- remaining instructions
val0 <= w;
ret0 <= true ;
end process message_0;

message_1 : process is
begin
wait until cal1 = true;
cal1 <= false;
-- instructions
cal2 <= true;
wait until ret2 = true;
ret2 <= false;
-- remaining instructions
val1 <= z;
ret1 <= true;
end process message_1;

message_2 : process is
begin
-- code
end process message_2;

message_3 : process is
variable temp : integer;
begin
wait until cal3 = true;
cal3 <= false;
-- instructions
if temp = 1 then
cal4 <= true;
arg4 <= temp;
wait until ret4 = true;
ret4 <= false;
end if
-- remaining instructions
ret3 <= true;
end process message_3;

message_4 : process is
-- code
end process message_4;

message_5 : process is
begin
-- code
end process message_5;
end architecture system;
```

The VHDL structural style is obtained by considering each process as a separate entity as well as all children sequence diagrams. For the sake of space, we do not show all messages VHDL code, rather than, we give the VHDL code only for *message_0*.

```
entity  message0 is
port (cal0 : inout boolean, cal1: out  boolean; ret0 : out boolean, ret1: inout boolean; arg0 : in integer; val0 : out integer; val1 : in integer);
end entity message0;

architecture basic of message0 is
begin
message_0 : process is
variable arg, val : integer;
begin
wait until cal0 = true;
cal0 <= false;
arg := arg0;
-- instructions
cal1 <= true ;
wait until ret1 = true ;
ret1 <= false ;
val:= val1;
-- remaining instructions
val0 <= w;
ret0 <= true ;
end process message_0;
end architecture basis;

entity  usecase_1 is
port (cal4 : inout boolean; arg4 : in integer; ret4 : out boolean);
end entity usecase_1;

architecture struct of usecase_1 is
signal cal2, cal5, ret2, ret5 : boolean
signal val5 : integer;
begin
```

```
messag2 : entity work.message2(basic)
      port map (cal2,ret2);
messag4 : entity work.message4(basic)
      port map (cal4,cal5,ret4,ret5,arg4,val5);
messag5 : entity work.message5(basic)
      port map (cal5,cal2,ret5, ret2, val5);
end architecture struct;

architecture struct of usecase_0 is
signal ret1, cal1, cal2, ret2, cal4, ret4 : boolean;
signal arg4, val1 : integer;
begin
messag0 : entity work.message0(basic)
      port map (cal0, cal1, ret0, ret1,arg0, val0, val1);
messag1 : entity work.message1(basic)
      port map (cal1,ret2,ret1,cal2, val1);
messag2 : entity work.message2(basic)
      port map (cal2,ret2);
messag3 : entity work.message3(basic)
      port map (cal3, cal4, ret4,ret3, arg4);
usecase1: entity work.usecase_1(struct)
      port map (cal4,arg4,ret4);
end architecture struct;

entity  test_bench is
end entity test_bench;
architecture test_usecase_0 of test_bench is
signal cal0, cal3, ret0, ret3 : boolean;
signal arg0, val0 : integer;
begin
usecase0 : entity work.usecase_0(struct)
        port map(cal0, ret0, arg0, val0, cal3, ret3) ;
stimulus : process is
begin
cal0 <= true ;
ret0 <= false;
arg0 <= 500;
val0 <= 0;
cal3 <= true ;
ret3 <= true ;
end process stimulus;
end architecture test_usecase_0;
```

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 is as follows:

```
// 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);}
```

### 4.3 Second technique

In this technique, we consider each end-to-end scenario as a VHDL process (SystemC SC_THREAD). An end-to-end scenario is a sequence of methods that are invoked by an external call from the environment. In this case, all processes communicate via shared variables. Table 2 shows the correspondence between UML and VHDL/SystemC concepts. All internal methods are implemented as VHDL procedures or functions. Since the same method may be called by many processes, we have to declare such methods globally in a VHDL package. We create ports only for external calls coming or returned values to the environment.

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

| UML concept | VHDL (behavioral /structural) | SystemC |
|---|---|---|
| End to end scenario | Process/Entity | SC_THREAD |
| Internal message without returned value | procedure | C++ function |
| Internal message with a returned value | function | C++ function |
| External call | port | port |
| Top level model | Test bench | sc_main() |

By applying this technique on the above example, we obtain two VHDL processes: *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*. We observe that *message_2* is called by both *process_1* and *process_2*. Thus *message_2* is declared globally in a package. We use the *use* clause to import all messages defined in the package. *work* designates the user library where are stocked files resulting from VHDL code simulation.

```
package pack is
procedure message_2;
```

```
end package pack;

package body pack is
procedure message_2 is
begin
-- message_2 body
end procedure message_2;
end package body pack;
```

The VHDL behavioral style for the two processes is as follows:

```
entity  usecase_0 is
port (cal0, cal3 : inout boolean; arg0 : in integer; ret0, ret3
: out  boolean; val0 : out integer);
end entity usecase_0;

architecture system of usecase_0 is
library work;
use work.pack.all;
begin
process1 : process is
function message_1 return integer is
variable result : integer;
begin
-- message_1 body
message_2; -- call to message_2;
-- remaining instructions
return result;
end function message_1;

function message_0(arg : in integer) return integer is
variable ret1, result : integer;
begin
-- message_0 body
ret1 = message_1; -- call to message_1
return result;
end function message_0;
-- process code
variable arg;
begin
wait until cal0 = true;
cal0 <= false;
arg := arg0;
val0 <= message_0(arg)
ret0 <= true;
end process process1;

process2 : process is
function message_5 return integer is
variable result : integer;
begin
-- message_5 body
message_2; -- call to message_2;
-- remaining instructions
return result;
end function message_5;

procedure message_4 (arg : in integer) is
variable result : integer;
begin
-- message_4 body
Result := message_5; -- call to message_5;
-- remaining instructions
end procedure message_4;

procedure message_3 is
variable result arg : integer;
begin
-- message_3 body
arg := arg4;
result := message_4(); -- call to message_4;
-- remaining instructions
end procedure message_3;
begin
-- process code
begin
wait until cal3 = true;
cal3 <= false;
message_3;
ret3 <= true;
end process process2;
end architecture;
```

The VHDL structural style for the two processes is as follows:

```
entity proc1 is
port (cal0 : in boolean; arg0 : in integer; ret0 : out
boolean; val0 : out integer);
end entity proc1;
architecture basic of proc1 is
begin
process1 : process is
-- process1 body
end process process1;
end architecture basis;

entity proc2 is
port (cal3 : in boolean; ret3 : out  boolean );
end entity proc2;
architecture basic of proc2 is
begin
process2 : process is
-- process2 body
end process process1;
end architecture basis;

architecture struct of usecase_0 is
begin
proces0 : entity work.proc0(basic)
    port map (cal0,arg0, ret0,val0);
proces1 : entity work.proc2(basic)
    port map (cal3,ret3);
end architecture struct;
```

The test bench architecture is the same as in the first technique. 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); }
```

### 4.4 Third technique

In this technique, each UML object is considered as a VHDL (SC_THREAD) process. 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 VHDL/SystemC concepts.

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

| UML concept | VHDL (behavioral /structural) | SystemC |
|---|---|---|
| Object | Process/Entity | SC_THREAD |
| Input message call | Input ports | Input ports |
| Output message call | Output ports | Output ports |
| External call | port | port |
| Top level model | Test bench | sc_main() |

By applying this technique on the above example, we obtain four processes (4): *Env*, *class_0*, *class_1*, and *class_2*. For the sake of the space, we give only the VHDL code for *Env* and *class_0*.

```
entity  usecase_0 is
port (cal0, cal1, cal2, cal3, cal4, cal5 : inout boolean; arg0, arg4 : inout integer; ret0, ret1, ret2, ret3, ret4, ret5 : inout boolean; val0, val1, val5 : inout integer);
end entity usecase_0;
```

```
architecture system of usecase_0 is
begin
Env : process is
variable temp : integer;
begin
cal0 <= true;
arg0 <= 1;
wait until ret0 = true;
ret0 <= false;
temp := val0;
--code
cal3 <= true;
wait until ret3 = true;
ret3 <= false;
-- remaining code
end process Env;

class_0 : process is
variable  arg, temp : integer;
begin
wait until cal0 = true;
cal0 <= false;
arg := arg0;
-- message0 instructions
cal1 <= true;
wait until ret1 = true;
ret1 <= false;
-- remaining message_0 instructions
ret0 <= true;
val0 <= w;
wait until cal3 = true;
cal3 <= false;
-- message3 instructions
temp := a;
if temp = 1 then
cal4 <= true;
wait until ret4 = true;
ret4 <= false;
end if
-- remaining message_3 instructions
ret3 <= true;
end process class_0;
end architecture system;
```

For the sake of space, we show only the structure of the *Env* process:

```
entity  Environment is
port (cal0, cal3 : out boolean; ret0, ret3 : inout  boolean; arg0 : out integer; val0 : in integer);
end entity Environment;

architecture basic of Environment is
begin
Env : process is
-- Env process code
end process Env;
end architecture basic;

architecture struct of usecase_0 is
signal cal0, cal1, cal2, cal3, cal4, cal5 : boolean;
signal arg0, arg4, val0, val1, val5 : integer;
begin
Envr : entity work.Environment(basic)
    port map (cal0, cal3, ret0, ret3, arg0, val0);
clas0 : entity work.class0(basic)
    port map (cal0, cal1, cal3, cal4, ret0, ret1, ret3, ret4, arg0, arg4, val0, val1);
clas1 : entity work.class1(basic)
    port map (cal1, cal2, cal4, cal5, ret1, ret2, ret4, ret5, arg4, val1, val5);
clas2 : entity work.class2(basic)
    port map (cal2, cal5, ret2, ret5, val5);
end architecture struct;
```

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) ;}
```

Table 4 compares between the three techniques.

**Table 4.** Comparison between the three techniques

| Technique | Processes Number | Process Granularity | Communication scheme |
|---|---|---|---|
| First | 6 | Fine | Message Passing |
| Second | 2 | Coarse | Shared memory |
| Third | 4 | Coarse | Mix |

### 4.5 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 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…figure 6 shows an example of an UML activity diagram with an action including three assignments written in AL, a call to a message called *Message_1* belonging to *class_0*, a condition, and a termination state. Note that in our case, only a subset of the AL is used. For instance, pointers are not used since existing Hardware synthesize tools do not know synthesize pointers to hardware. Instead of, we use arrays. VHDL supports a large set of operators and control instructions found in AL. Using the Rhapsody environment we can perform functional simulation before HDL code generation. This step is very important in order to validate the HDL code functionality against UML functional models.

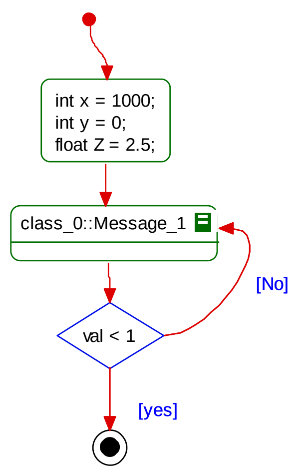



**Figure 6.** Example of UML activity diagram

## 5. Implementation and case study

We have used the Rhapsody environment for UML modelling and HDL code generation. In order to automate the VHDL/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 VHDL/SystemC files (see figure 7). We have developed two VB macros for SystemC/VHDL codes generation and integrated them as tool boxes in the Rhapsody environment. As a case study, we have chosen the SDP (Simplex Data Protocol) [19] application whose UML sequence diagrams are illustrated in figure 8. Figure 9 shows the UML activity diagram for the receiver object. Figure 10 gives us an overview of SystemC files for the *receiver* object.

## 6. Conclusion

In this paper, we present our approach for automatic VHDL/SystemC code generation from UML models at early stages of embedded systems development. Our proposed flow consists mainly of two steps: generation of VHDL/SystemC codes from UML hierarchic sequence diagrams then from UML activity diagrams. The generated VHDL/SystemC code at the first stage is used for algorithmic space exploration and simulation purposes using existing commercial simulators. In the second step, we introduce UML activity diagrams to model messages internal behaviours. Actions of activity diagrams are expressed in the C++ Action Language (AL) which is included in the Rhapsody environment. From AL, a full VHDL/SystemC code is generated for both simulation and synthesis. VHDL/SystemC code is generated as text files automatically and this is due to the VB API included in the Rhapsody environment. In order to enable designer to explore the algorithmic space, we proposed three techniques for HDL code generation. According to simulation results, the designer can restructure his/her system by increasing or decreasing the processes number (i.e. merge or scatter processes). As a perspective, we plan to investigate the MDA approach for VHDL/SystemC code generation from sequence diagrams and consider asynchronous events and temporal constraints.

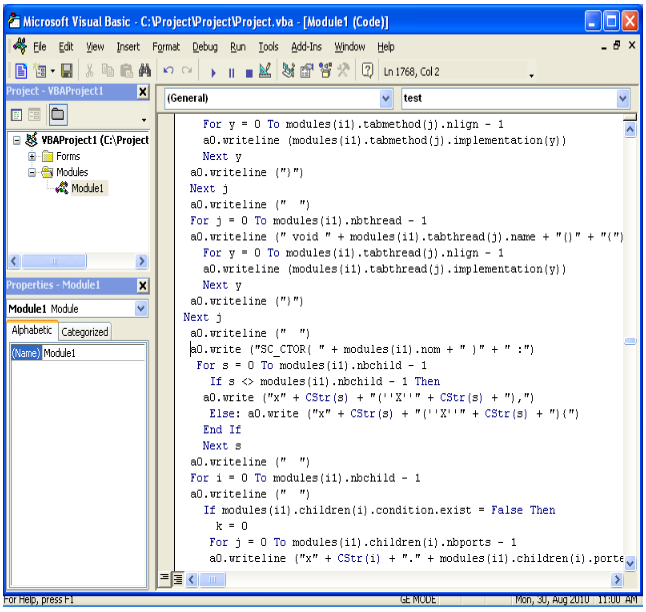
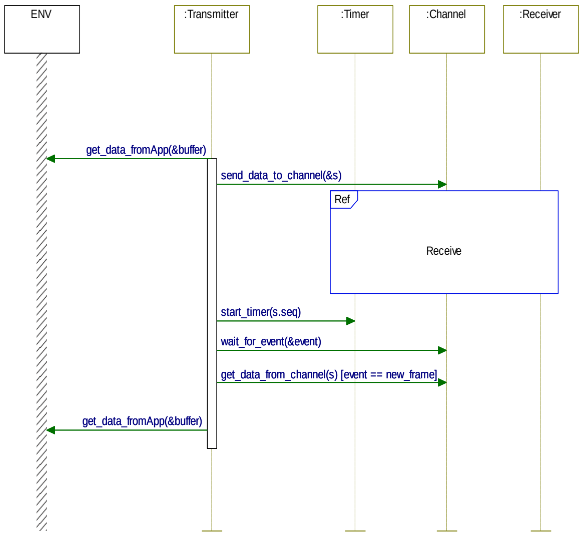
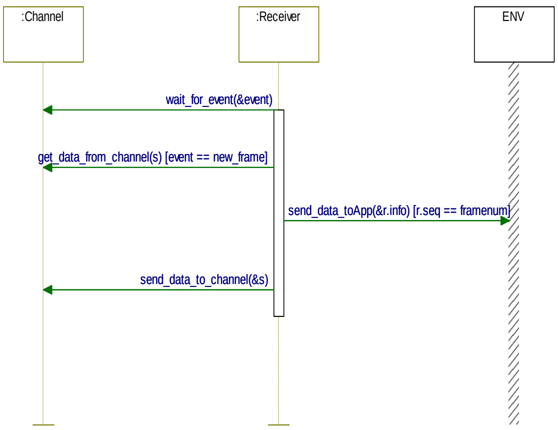



**Figure 7.** Programming with VB API




**(a)**




**(b)**

**Figure 8.** UML sequence diagrams for SDP

(a) Main sequence diagram; (b) sequence diagram for receive use case.

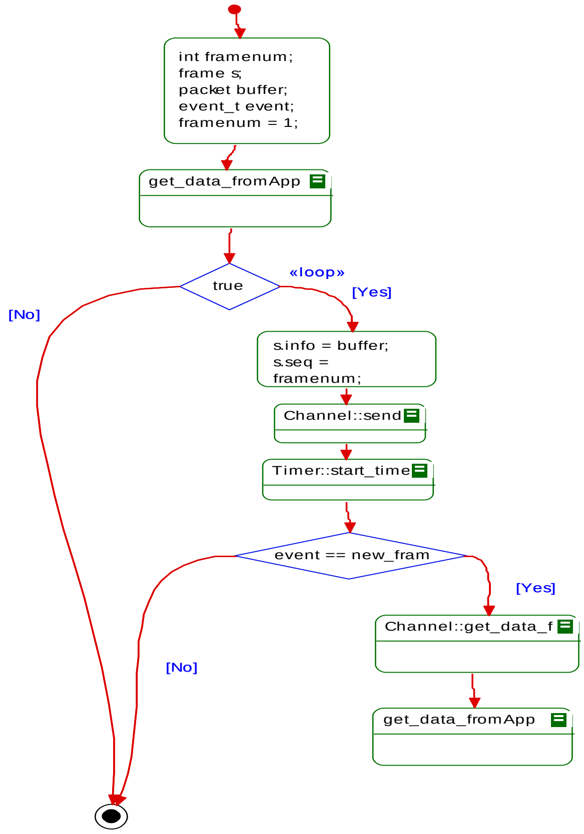



**Figure 9.** UML activity diagram for Receiver object

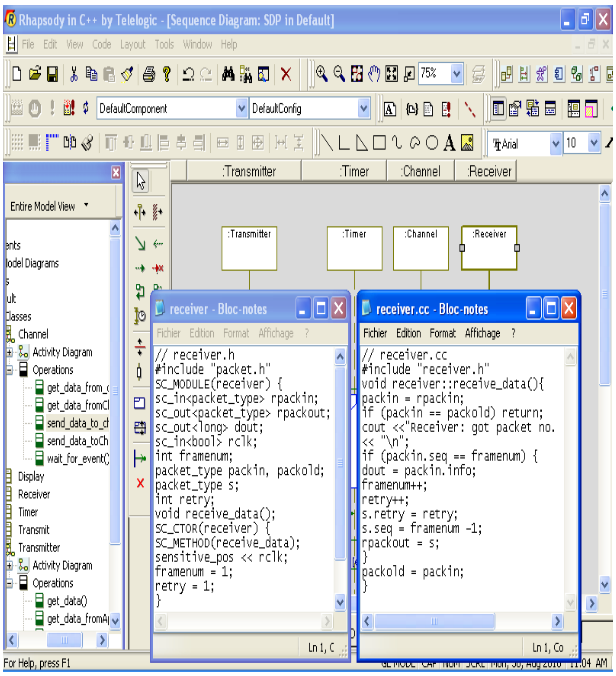



**Figure 10.** SystemC code generation from Rhapsody UML models

## References


[1] Akehurst, D.H., Uzenkov, O., Howells, W.G., Mcdonald Maier, K.D., Bordbar, B., “Compiling UML state diagrams into VHDL: An experiment in Using Model Driven Development”, FDL’07, 2007.

[2] Ashenden, P.J., “The VHDL cookbook”, first edition, published by Morgan Kaufmann, 1990.

[3] Ashenden, P.J., “VHDL Tutorial”, Elsevier Science (USA), 2004.

[4] Bjarklund, D., Lilius, J., Poress, I., “Towards efficient code synthesis from Statecharts”, Puml Workshop at UML2001, 2001.

[5] Bjarklund, D., Lilius, J., “From UML behavioral descriptions to efficient synthesizable VHDL”, proceedings of 20th IEEE NORCHIP Conference, Copenhagen, Denmark, 2002.

[6] Booch, G., Rumbaugh, J., Jacobson I., “Unified Modeling Language User Guide”, Addison-Wesley, 1999.

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

[8] Coyle, F.P, Thornton, M.A., “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, 2005.

[9] Gajski, D., Vahid, F., Narayan, S., Gong, J., “Specification and Design of Embedded Systems”, Prentice Hall. Englewood, New jersey 07632, 1994.

[10] Jerraya, A.A., Wolf, W., “Multiprocessor systems on chip”, Morgan Kaufmann publishers, 2005.

[11] Martin G., “UML for embedded systems specification and design: motivation and overview”, Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings, p. 773–775, 2002.

[12] McUmber, W.E., Cheng, B.H.C., “UML-based analysis of embedded systems using a mapping to VHDL”, proceedings of IEEE Int. Symposium on High Assurance Software Engineering (HASE’99), Washington, DC, USA, p. 56-63, 1999.

[13] ModelSim documentation, ftp://ftp.xilinx.com/pub/documentation.

[14] Narayan, S., Vahid, F., Gajski, D.D, “Translating system specifications to VHDL”, IEEE European Design Automation Conference, Amsterdam, Netherlands, 1991.

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

[16] Riccobene, E., Scandura, P., Rosti, A., Bocchino, S., “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), 2005.

[17] Rieder, M., Steiner, R., Berhouzoz, C., Corthay, F., Sterren, T., "Synthesized UML, a Practical Approach to Map UML to VHDL", LNCS, Volume 3943, 2006.
[18] SystemC, Functional specification for SystemC 2.0, www.systemc.org, 2002.
[19] SystemC, Version 2.0 User's guide, www.systemc.org, 2002.
[20] SystemC, IEEE Standard SystemC® language Reference Manual, www.systemc.org, 2005.
[21] Thomson, R., Chouliaras, V., Mulvaney, D., Plessis, P., "From UML to Structural Hardware Designs", UMLSOC, 2007.
[22] UML2.0 Superstructure Specification, http://www.omg.org, 2003.
[23] VHDL, IEEE Standard VHDL Language Reference Manual. IEEE, IEEE Std 1076, 2000.
[24] Vidal, J., De Lamotte, F., Gogniat, G., Soulard, P., Diguet, JP., "A codesign approach for embedded system modeling and code generation with UML and MARTE", DATE09, 2009.

## Author Biography

**Fateh Boutekkouk** was born in Constantine (Algeria). He received his BS degree in Computer science from the University of Constantine, his MS degree from the University of Jijel (Algeria), and his PhD from the University of Constantine in 2010. He is an assistant professor at the University of Oum el Bouaghi (Algeria) since 2003. His current research interests include Software Engineering, Embedded Systems and Systems On Chip (SOC) design.