Synthesized UML, a Practical Approach to Map UML to VHDL

Medard Rieder, Rico Steiner, Cathy Berthouzoz, Francois Corthay, and Thomas Sterren

Infotronics Unit (www.infotronics.ch),
University of applied Sciences Valais (www. hevs.ch),
Rte du Rawyl 47, 1950 Sion, Switzerland
{rim, sti, bet, cof, sth}@hevs.ch

Abstract. Embedded Systems are complex systems with limited resources such as reduced processor power or relatively small amounts of memory and so on. The real time aspect may also play an important role, but is definitely not a main consideration of this work. Complexity of recent embedded systems is growing as rapidly as the demand for such systems and only can be managed by the use of a model-driven design approach. Since modeling languages such as UML are semi-formal they allow the design of systems that can't be implemented using formal languages such as C/C++ or VHDL. This paper intends to show how the gap between model and formal language can be bridged. First of all a set of rules restricts the use of model elements in a way that the model will become executable. Furthermore a unique mapping between UML and formal language elements enables automatic code generation. Formal verification at model level is an important consideration and becomes possible by the fact that rules restrict the application of model elements. UML to software (C/C++) and UML to hardware (VHDL) mapping form the base for a practical codesign approach where a part of the system is realized through software and another part trough hardware. Mapping of UML to programming languages is well known today and realized in many tools. Mapping of UML to hardware description languages is less known and not realized in tools. This paper documents an attempt to define a set of rules and to implement UML to VHDL mapping in a practical code generator. It also shows parts of a real world sample that was realized to verify usability and stability of rules and mapping. Finally, an outlook on further developments, improvement of the UML to VHDL mapping and a simple codesign process called 6qx will be given.

1 Introduction

While Embedded Systems were not widespread before 1990, nowadays they have become very popular. Affordable prices of big sized memory and powerful processors form the ideal alchemy for the birth of numerous embedded systems. Another component of this alchemy is the fact that hardware has become programmable. Field Programmable Gate Arrays (FPGA) with sufficient number of gates at reasonable prices made the borderline between hardware and software vanish.

Even though there are a rising number of basic components for embedded systems, and new technologies appear in rapid succession, the system development cycle is still quite traditional as illustrated by figure 1.

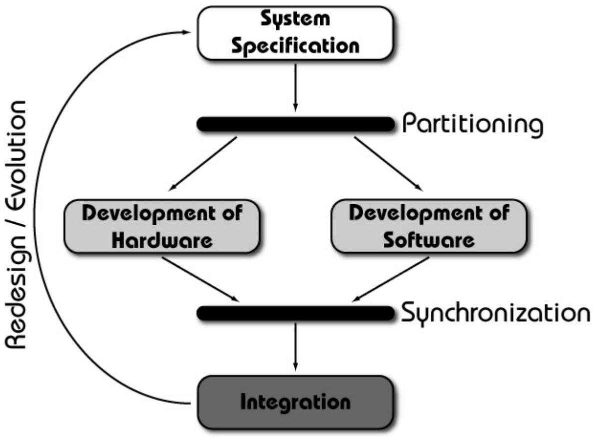

Fig. 1. Traditional Embedded System Development Cycle

2 Codesign

In such a traditional process, hardware and software are developed parallel, which brings up several issues, such as:

- Need of early hardware and software partitioning.
- Asynchronous development of hardware and software.
- Late integration with possible need of redesign.
- Missing hardware prevents testing the software before integration.

An important reason why development of hardware and software is not integrated is the lack of simple model-based approaches. Several reasons prevent the use of model-driven development.

Existing Codesign tools are very expensive and mostly dedicated: If codesign tools exist, these are almost certainly very expensive and dedicated to a specific thematic and platform. Readapting to other thematic and /or other platforms is practically impossible.

Developers think in terms of code and not model: Traditional thinking [1, 2, 3] and often also investments that have been made into some existing platform inhibit a change of attitude. Since formal descriptions are what they are and do not heal lack of methodic approach, first experiences in modeling are mostly disappointing and many hardware and software programmers therefore fall back into well-known territory, which means thinking either in hardware or in software code.

Therefore the HEVs approach of a codesign method was to use existing software modeling techniques already established on the market and to bridge the gap between software engineering and system engineering (codesign) by adding the hardware engineering part. How this was done will be described in detail in the following chapters and sections.

2.1 A Theoretical Codesign Approach

As a theoretical approach, we have developed a quite simple pyramid with the integrated system model as its top.

As underlying layer, we split up the model into a hardware model and a software model section. This process is called partitioning. The partitioning is done manually to give us the most flexibility to draw the boarder line (Figure 3) between hardware and software. However all needed interfaces between hardware and software are automatically created. Each of the models will then be translated into either hardware code (VHDL) [4] or software code (C/C++) [5]. Afterwards, the code will get synthesized or compiled and then uploaded into the target system. These last two steps are automated and require no user interaction. Figure 2 shows the theoretical model.

Finally there has to be a formal verification step. The produced code has to be verified against the model. Not only the software and the hardware code have to be verified, also the semantics of the overall system have to be tested. Timing constraints has to be tested to.

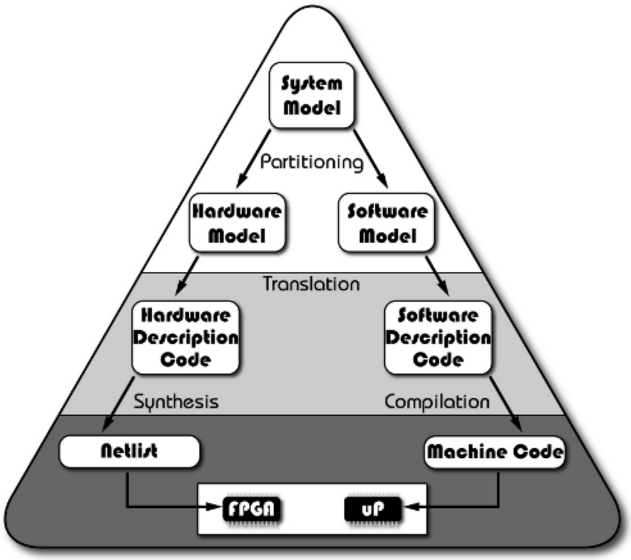

Fig. 2. Model Driven Codesign of Embedded Systems

It can easily be seen that integration problems will be minimized, since integration is already part of the model. It also can be seen that different degrees of partitioning are possible throughout this model. Figure 3 shows both, one hardware (a) and one software centric (b) partition (solution) of a given system.

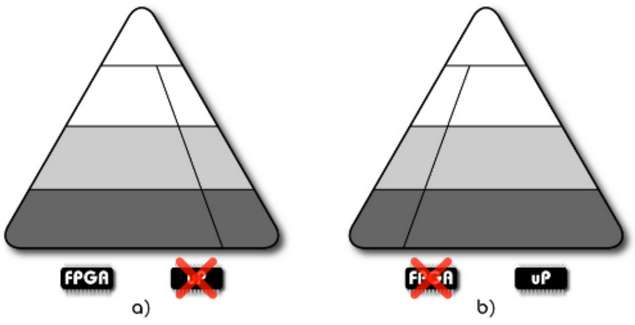

Fig. 3. Different Degrees of Partitioning

2.2 A Practical Codesign Approach

Theoretical approaches are nice to have a basic understanding, but to come to true results, practical models have to be developed out of the theoretical ones. We did this by instantiating a codesign model using realistic tools and targets. Figure 4 shows an overview of this practical approach.

To make complexity of this problem reasonable some constraints are introduced:

- Actually, we can do the two extreme partitions: either full hardware or full software.
- A formal verification of the produced code against the model is not yet possible.
- Real time aspects are only partially taken into account. The system has to reproduce the behavior specified by the model. The code is not able to handle hard-real-time situations. But it is possible to generate very compact and target specific code due to the flexible translation mechanism.

To understand figure 4, one must understand the UML approach we use to manage different partitions of the embedded system on model level. Packages, classes and state charts are used to model the target independent elements and the behavior of the system. Further on one component is defined for each of the partitions (hardware / software). Interfaces allow using the same system description for several partitions and targets. Doing so makes it possible to define an arbitrary number of hardware respectively software components for an arbitrary number of targets. A partition specific component holds all the information that is model-level related such as which packages or classes are part of this specific partition. It also holds all target specific information such as which tool will generate the code, how this code will be translated, how it will be synthesized or compiled and how it will be uploaded to the embedded system. In this way it is possible to automate the entire build and execute command chain.

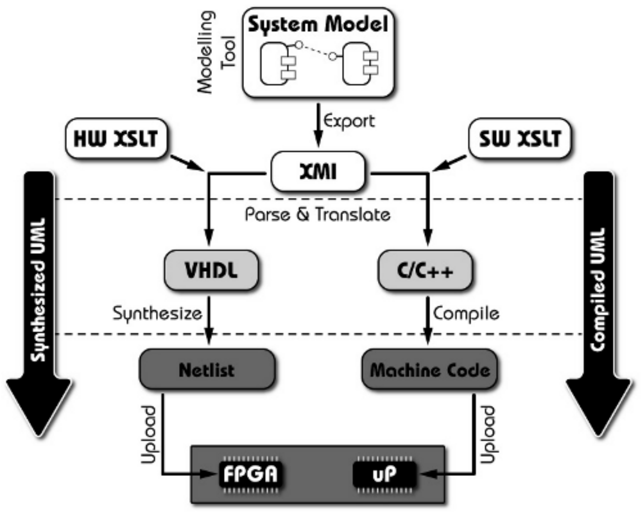

Fig. 4. A Practical Codesign Approach

Basically, the build command is run with the components name as parameter, the entire model will then be exported and either the hardware or software translator parses and translates information related to the specific component (partition) out of the exported model. It would also be possible to have a single translator, which receives one more parameter that determines whether to translate model information into hardware or software code. For reasons of simplicity (a translator is a quite complex matter) we decided to build two separated ones.

It has to be emphasized that the partitioning is done manually by defining components and assigning packages and classes to them. Also, components have to be equipped with target specific information. But it has also to be emphasized that partitioning is done after modeling the system and just before generating the code.

3 Translation

Correct translation of the UML elements into code is the core problem of any realistic codesign approach. Many researchers have already worked on this problem. The translation of UML to software code has been thoroughly researched and today offers good stability and performance. When it comes to translation of UML to hardware the papers of McUmber & Cheng [6, 7] are good examples. Unfortunately many of these works lead towards code that can't be compiled/synthesized because they have their main focus on the model level. For our work it is a main consideration that the generated code has to be compileable/synthesizable. To do real codesign, both, model and generated code have to be adapted to each other. Therefore the main effort of our research went towards this problem [8]. The next sections describe the main results of this work.

3.1 Hardware Thinks Differently

To communicate, using events is very common in UML. But the concept of events known from software doesn't exist in hardware. The one and only event in hardware is the continuous clock signal, the system clock. All other communication is done using signals that hold their value until they are told to change it.

In software, events can be used for communication tasks. But in hardware that isn't true. Even if one defines a signal with a pulse width of one period of the system clock as an event, this won't be an event, because only the value of the signal is taken in consideration and not the pulse width. In UML this would mean that state transitions are only decorated with a guard and no trigger.

This, and the fact that UML is closely related to software, brings us to filling the gap between UML and hardware. Therefore it's needed to develop a communication mechanism that can act as expected in UML (see section 3.3). Further on we need do define some rules, which coordinate the use of UML for hardware and software systems. There are three reasons to do this:

- In every Hardware Description Language (HDL) one can describe functions and situations that can't be synthesized. But if the designer follows some basic and simple rules, he can be sure that the design is synthesizable. In UML, the same situation exists. One can design a model, which can be translated neither to software nor to hardware.
- Till now, UML [9] was used to design software [10, 11] only. There is a lack of experience when it comes to creating models that can be translated to hard- and software. Defining some rules will help the designers to improve their know-how and it will bring a certain amount of quality to a model.
- Guided by these rules, the designer can be sure that a model is suitable for software/hardware codesign.

Above-mentioned rules would normally be part of a hardware/software process. Currently we are working on such a process (see section 5.2) and describing these rules now would go beyond the scope of this document. Instead, we would like to concentrate on the mapping of UML elements to VHDL code, which is done in the next section.

3.2 UML Elements

Since there are a big number of elements in UML, not all of them have been taken into account for this first approach of the UML to VHDL translation. The elements taken are classes, attributes, operations, class diagrams, objects, object diagrams, associations, ports, interfaces, events and state charts. As our real world sample shows, these elements are sufficient to model the behavior of simple systems [6, 7].

Comparing these elements to the traditional concepts used by hardware designers shows a strong parallelism [10, 12, 13]. Using state charts became a very popular concept for hardware designers to describe a design before implementing it. The top-down concept for analyzing hardware designs is also widely used. The system is seen as a black box with inputs and outputs. The black box will be broken down in smaller parts seen as black boxes again. The inputs and outputs of the black boxes will be connected to establish communication. This process continues until a certain granularity is achieved and finally the black boxes will be equipped with a behavior described e.g. with a state chart. Using object diagrams, the same analyses and design process is possible while an object corresponds to a black box. Also communication between objects is possible by using ports and interfaces. An object can be equipped with ports and interfaces, and ports are interconnected by links. Figure 5 shows a typical sample.

The next section provides more details of the translation technique we used.

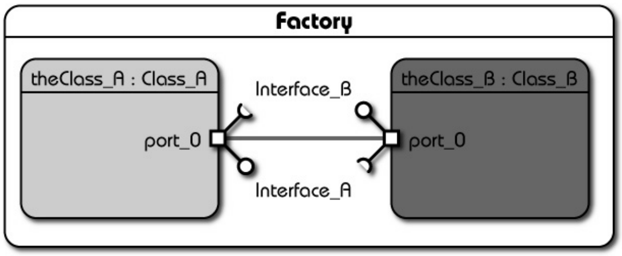

Fig. 5. UML Object Diagram

3.3 UML to VHDL Mapping

It's important to find optimal patterns to translate UML elements to VHDL. Due to the fact that an UML element can have several decorations, it is important to find a general description in VHDL that can handle all the decorations of an UML element. Not going too deep into details we will show now, which UML element looks how in VHDL. To give an idea of the translations result, figure 6 shows the VHDL code corresponding to the UML elements in figure 5. The following lines will give an overview and a brief description of the patterns used.

- The class *Factory* is translated to an **entity** construct. The entity named *Factory* consists of a list of **ports** and of an **architecture** section. By default the two inputs reset and clock are added to the entity's port list. Depending on the ports and interfaces of the class, other input and output ports may be added to the port list. In the **architecture** the main implementation (the behavior) of the class can be found.
- An object is an instance of a class. In VHDL it's possible to create an instance of an **entity**. To do this, first a **component** has do be defined inside the **architecture** containing the instance. According to figure 5 the **components** named *Class_A* and *Class_B* are defined (cf A:, B: at figure 6). The second step is to create an instance and to map the ports to other ports or to signals. The instances are called *theClass_A* and *theClass_B* (cf C:, D: at figure 6).

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Factory is
port(
  reset : in STD_LOGIC;
  clock : in STD_LOGIC
  -- other ports here.
  );
end Factory;

architecture FA of Factory is
---- Declare signals to
 interconnect nested blocks.
Signal
link_0_Interface_A_methode_0:
STD_LOGIC;
Signal
link_0_Interface_B_methode_0:
STD_LOGIC;

---- A: Declare class Class_A
component Class_A
port(
  clock : in STD_LOGIC;
  reset : in STD_LOGIC;
  port_0_Interface_A_methode
_0 : in STD_LOGIC;
  port_0_Interface_B_methode
_0 : out STD_LOGIC
  );
end component;

---- B: Declare class Class_B
component Class_B
port(
  clock : in STD_LOGIC;
  reset : in STD_LOGIC;
  port_0_Interface_B_methode
_0 : in STD_LOGIC;
  port_0_Interface_A_methode
_0 : out STD_LOGIC
  );
end component;

begin
---- C: Instantiate
theClass_A
theClass_A: Class_A
port map (
  reset => reset,
  clock => clock,
  port_0_Interface_A_methode
_0 =>
link_0_Interface_A_methode_0,
  port_0_Interface_B_methode
_0 =>
link_0_Interface_B_methode_0
  );

---- D: Instantiate
theClass_B
theClass_B: Class_B
port map (
  reset => reset,
  clock => clock,
  port_0_Interface_A_methode
_0 =>
link_0_Interface_A_methode_0,
  port_0_Interface_B_methode
_0 =>
link_0_Interface_B_methode_0
  );
end Factory
```

Fig. 6. Generated Code of UML Diagram in Figure 5

- The UML elements port and interface are translated to input and output ports of an **entity**. Provided interfaces are translated as input ports and required interfaces are translated as output ports. The names of VHDL ports are the same ones as the names of the ports and interfaces in the UML model (cf A:, B: at figure 6).
- In UML, links are used to interconnect objects via ports and interfaces (see link between the two objects in figure 5). Depending on the interfaces, ports and objects used, several signals will be defined in VHDL. These signals will be used in the port map sections of components to realize a connection between components (cf C:, D: at figure 6).

The second type of UML diagrams discussed here are state charts. Figure 7 shows a simple but complete state chart that is equipped with all common elements. Comparing the state chart in figure 7 to its implementation in VHDL (figure 8), we will explain the mapping rules for state charts.

As state charts are used to describe the behavior of a class, the corresponding translation is put into the **architecture** of the VHDL **entity**. State charts are translated into a case structure where every case represents a state of the state chart. Within every case an if-construction prevents reentrant execution of the state's embedded instructions. Another if-construction is used to handle the transitions and the transition conditions.

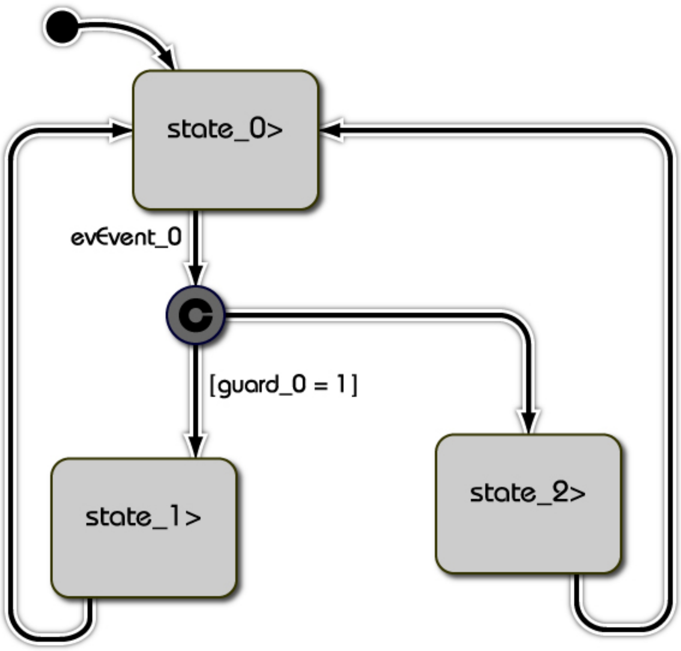

Fig. 7. UML State Chart

- The code in figure 8 shows a state chart called *Statemachine_0* within the Class *Class_A*. A state chart is realized within a VHDL **process** (cf A:, B: at figure 8). The process is triggered by the two signals reset and clock. The process consists of an a synchronous reset a part (cf C: figure 8) and of a main part synchronous to the clock (cf D: figure 8). If the reset signal goes to '1' then the circuit gets initialized and initial state is defined. The synchronous part implements the behavior of the state chart.
- Defining a new data **type** called *TFSM_States* does represent the set of states need to implement the state chart. Further on a **signal** called *FSM_Statemachine_0* is created, which type is *TFSM_States* (cf E: figure 8).
- The state chart its self is translated to a **case** structure (cf F: figure 8) that treats all possible values defined by the data type *TFSM_States*. Each state corresponds to a well-defined value of the **signal** *FSM_Statemachine_0*. A very special case or an undefined value of *FSM_Statemachine_0* is caught by the **when others** statement. This grants that the state chart does not get deadlocked and that it goes back to the initial state.

```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity Class_A is
port(
  reset : in STD_LOGIC;
  clock : in STD_LOGIC;
  evEvent_0_reception: in STD_LOGIC
);
end Class_A;

-- A: declare the class Class_A
architecture Cl_A of Class_A is

-- E: state chart type and signal
-- definition
type TFSM_States is
(
  S_state_0,
  S_state_1,
  S_state_2
);

signal FSM_Statemachine_0:
TFSM_States;

-- multiple action exe. prevention
signal enAction: STD_LOGIC;

-- Event Reception signal def.
signal evEvent_0: STD_LOGIC;

begin
-- B: process for Statemachine_0
  Statemachine_0:process(clock,
reset)
  Begin
    -- C: asynchrony reset
    if reset = '1' then
      FSM_Statemachine_0 <=
S_state_0;
      enAction <= '1';
-- D: synchrony part
    elsif RISING_EDGE(clock) then
-- F: state chart implementation
      case FSM_Statemachine_0 is
        when S_state_0 =>
-- H: multiple execution prevention
      if enAction = '1' then
        -- state’s operartions
        enAction <= '0';
      end if;

-- J: event handling
       if evEvent_0 = '1' then
-- K: guard handling
          if guard_0 = '1'then
-- G: state assignment
            FSM_Statemachine_0
<=
             S_state_1;
-- I: enable actions
            enAction <= '1';
          else
            FSM_Statemachine_0
<=
             S_state_2;

     enAction <= '1';
          end if;
       end if;
       when S_state_1 =>
          if enAction = '1' then
---- operartions of the state
            enAction <= '0';
          end if;

          FSM_Statemachine_0 <=
S_state_0;
            enAction <= '1';
       when S_state_2 =>
          if enAction = '1' then
---- operartions of the state
            enAction <= '0';
          end if;
          FSM_Statemachine_0 <=
S_state_0;
          enAction <= '1';
          end if;
       when others =>
          FSM_Statemachine_0 <=
S_state_0;
          end case;
       end if;
     end process;
       evEvent_0_ev_generator:
ev_generator
     port map (
       reset => reset,
       clock => clock,
       trigger =>
evEvent_0_reception,
       evOut => evEvent_0
     );
end Class_A
```

Fig. 8. Generated Code of UML Diagram in Figure 7

- Each state has one or more subsequent states. These subsequent states are assigned by statements like `FSM_Statemachine_0 <= s_state_1` (cf G: figure 8).
- If in UML a state is provided with an operation, it is assumed that this operation is executed only once. To reproduce this behavior in VHDL an additional signal

called *enAction* is introduced. After a reset enAction has the value ‘1’. Inside the **case** structure, the value of enAction is checked for each state. If enAction’s value is ‘1’ then the state-related actions are executed. That followed, enAction is set to ‘0’. Doing so prohibits execution of the state’s actions when reentering the same state (cf H: figure 8).

- In order to enable execution of action in case of going to another state, enAction has to be reset to ‘1’. This is done in the same moment as the **signal** FSM_Statemachine_0 becomes a new state (cf I: figure 8).
- As one can see in figure 7 UML defines several types of transitions. A transition can be decorated with *triggers* and *guards*. Triggers are events (generated from other parts of the model or timer events), which bring the system into another state. Guards are additional conditions that can allow or reject a state change in case of an occurring event. In this example occurring of the event evEvent_0 means that the **signal** evEvent_0 becomes ‘1’ for exactly one clock period. If this is the case the guards are evaluated and the next state is defined as explained above (cf J: figure 8).
- Guards are realized in a simple **If** structure (cf K: figure 8). If the transition’s guard is evaluated and the result is true, the system goes to the transition’s target state. There can always be maximum one transition of the same source state without guard. If all transition’s guards are evaluated as false, the system goes to the target state of the guardless transition.

As mentioned in section 3.1 we had to establish a communication mechanism between components. This necessity comes from the fact that within e.g. a state we can have a method call. To keep the generated VHDL code human-readable we avoid changing the anatomy of the case structure representing the state chart. To introduce a pseudo-event to translate e.g. method calls we defined a communication mechanism as shown in figure 9.

- Generating an event means changing a signals value from ‘0’ to ‘1’ or from ‘1’ to ‘0’.
- Receiving an event is as simple as to detect a signal’s value changing.
- If the event transmitting signal is initialized properly, throwing an event is as simple as inverting the signal’s value. Detecting an event is slightly more complex. It’s managed by a simple edge detection mechanism. As soon as the event’s signal is changed, an output signal changes from ‘0’ to ‘1’ for exactly one period of the system clock. Figure 9 shows the different signals as mentioned above. This is not the best approach to solve the problem. E.g. sending two events to the same receiver is not possible and quickly occurring events could be missed.

For the hardware mapping we use basic elements of VHDL and well-known structures. This has a number of advantages:

- If the generated code is semantically correct, it is granted that the generated code is synthesizable.
- The generated code is platform and manufacturer independent. This is because we don’t use target specific elements such as memory or multiplier blocks.

The next chapter describes a demonstrator that was created to verify the techniques exposed in this chapter.

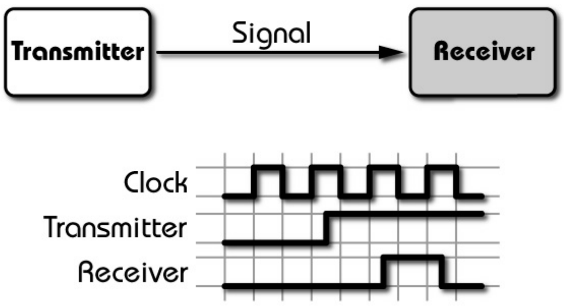

Fig. 9. Communication Mechanism to handle Events

4 Experimentation

As usual in research projects, at the end all obtained results have to be verified and proven.

A simple chronometer demonstrator was built up. As software target, we used an ARM 7 equipped board with a minimal operating system called IDF (Interrupt Driven Framework) and as hardware target a Xilinx Spartan II equipped board. The chronometer itself consists of a stepper motor, some pushbuttons and an optical sensor. An UML model of the system was created and then once synthesized for the hardware target and once compiled for the software target. Both systems were working without touching the model. All code was automatically generated, compiled, uploaded and started in the targets. Figure 10 shows the schematic of the demonstrator.

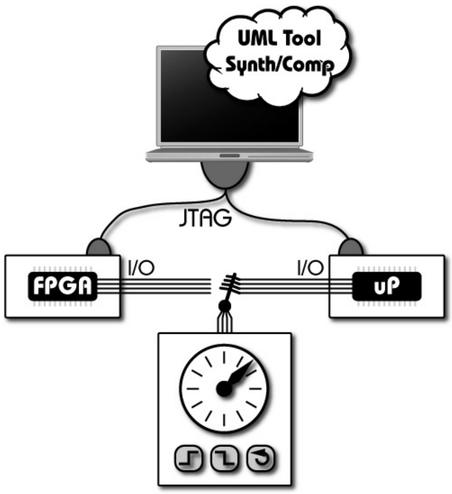

Fig. 10. Principle of the Chronometer Codesign Demonstrator

The following conclusions could be made out of the chronometer experience:

- The software code generator we used, was the built-in one of the modeling tool. For future development of the codesign project, it has to be replaced by one of the same types as the hardware code generator. This is necessary to allow correct interface integration between hardware and software.
- Moving the partition line in between of hardware and software means to involve interfaces. For this time, we implemented interfaces manually but for a real world development process it will be a must to at least semi-automate this action. This means that it should be possible to "drag & drop" ready-made interface blocks into the model and to connect them to the correct locations in the hardware and software fragment of the system's model.
- The mapping for the hardware should also be optimized. Especially it would be nice to be able to parameterize target specific matter inside the model and not to find these adaptations somewhere in the translator.
- Not all elements of UML 2.0 [9] have been used. This was partially due to the fact that the software built-in translator did not recognize them, at least not in the version of the tools we were using. This problem will automatically be corrected by introducing the "homebrew" software translator.
- More complex demonstrators must be implemented to stress- test the codesign approach we are using currently. But it must be stated that the results we obtained until now are very encouraging and that the generated systems are amazingly stable.

Having a frame, inside of which development is rolling down, would be nice. A first approach of such an all-over Development Process is briefly touched in the conclusion.

5 Conclusion

The above-mentioned experiences lead to a certain number of conclusions that have to be applied in the very near future to the described codesign approach.

5.1 Tool Chain Improvement

The most important improvements concerning the tool chain are shown in the following list:

- Improved hardware code generation patterns will be implemented in the hardware translator.
- A separate software translator will be added to the tool chain.
- Standard interfaces will be defined in UML as patterns that can be applied to a given situation.
- The whole approach will be intensively tested by the means of real world projects and demonstrators.

These are all developments that will be done around the tool chain. Another, more important development, will be to introduce a general formalism that embeds the present experimental codesign process. The reason for this is that any modeling activity requests formalism and sequencing. An outlook on this process is presented in the next section.

5.2 The 6qx Process

The definition of a simple codesign process is the logical consequence of this conclusion and because we already had defined a software centric process (6q) [14], we will extend this one into a codesign process. The 6q process has been developed in an embedded systems context and therefore provides a quite good potential to cover also hardware development aspects. The method of the 6q process is object oriented and the model is incremental. It consists of six major steps: System specification, analysis, design, translation, validation and integration.

These steps will also be contained in the new 6qx process, but will be adopted to meet codesign requirements as follows:

The first two steps, system specification and analysis, gather information about the system to be developed and map results into an UML model by the means of use-case diagrams, interaction diagrams, class diagrams, state charts and deployment diagrams. Since these steps try to specify and analyze the system, they do not care about implementation details (hardware / software). The major difference of these steps compared to the original 6q process, where hardware and software are developed in parallel, will be the removal of the hardware software partitioning decision. It is delayed into the design step, because the model covers both hardware and software of the system.

The design step will transform the flat analysis model into a well-structured model that can be partitioned into a hardware (HW) and a software (SW) partition according to various criteria (costs, speed, physical limitations and so on). The hardware partition may be split up once again into a programmable hardware (PHW) partition and an analog / digital hardware partition (DHW). Interfaces between both partitions have to be defined after partitioning or even while partitioning. The 6qx process will contain recommendations about use and implementation of interfaces in the form of interface patterns defined in UML. It will also contain hardware and software design rules (cf. section 3.1) in form of patterns defined in UML.

The translation step will regulate implementation details. Important elements that will be introduced at this moment into the system model are components that will bind the hardware and the software partition to specific targets (cf. section 3.2).

The validation step is responsible to verify correct functioning of the designed hardware and/or software. This is achieved by reusing formal descriptions of behavior from the specification and analysis step. Simulators are used to verify correct behavior.

The integration step will put it all together and finally verify correct all-over system behavior and stability. Erroneous behavior results in feedback towards the analysis pipe, insufficient stability in feedback towards the design pipe.

Figure 11 gives an idea of the 6qx process.

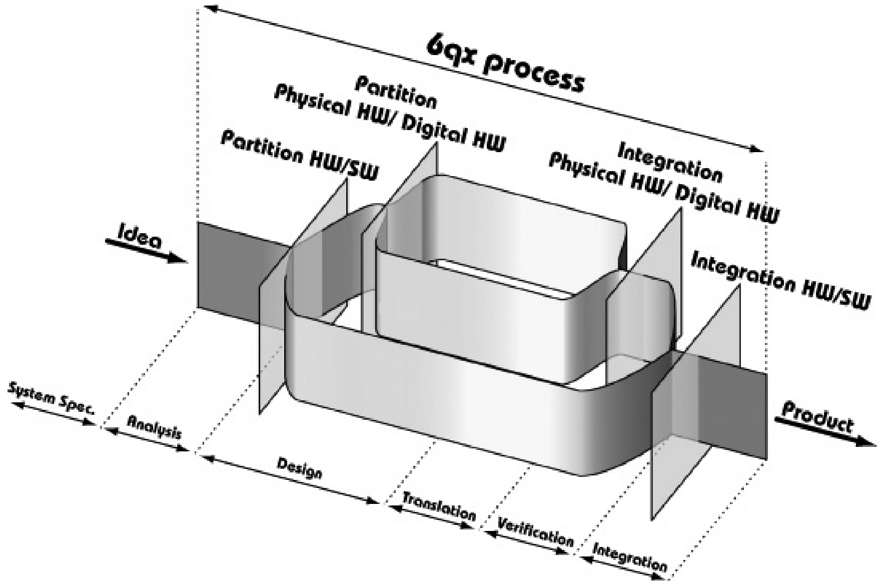

Fig. 11. Overview of the 6qx codesign process

References

[1] Keith Edwards, "Real Time Structured Methods", John Wiley & Sons, 1993.

[2] Jean J. Labrosse, "Embedded Systems Building Blocks", R&D Publications, 1995.

[3] D. L. Perry, "VHDL", second edition, McGraw-Hill, New York, 1994.

[4] IEEE, "IEE Standards VHSL Language Reference Manual", IEEE standard 1076-1987, IEEE Computer Society Press, Los Alamitos, 1987.

[5] Marylène Micheloud, Medard Rieder, "Programmation orientée objets en C++", Presses polytechniques et universitaires Romandes, 2003.

[6] McUmber & Cheng, "A general Framework for formalizing UML with Formal Languages", ICSE 2001.

[7] W. E. McUmber and B. H. Cheng, "UML-based analysis of embedded systems using a mapping to VHDL", In Proc. of IEEE High Assurance Software Engineering (HASE99), Washington, DC, November 1999.

[8] Rico Steiner, "Hardware & Software Co-Design", Diploma Thesis, HEVs, 2004.

[9] www.uml.org.

[10] Bruce Powel Douglass, "Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems", Addison-Wesley, 2004.

[11] Bruce Powel Douglass, "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns", Addison-Wesley, 1999.

[12] S. Jones, D. Naylor, "VHDL, a logic synthesis approach", Chapman & Hall, London, 1995.

[13] Rajan "Essentials VHDL, RTL synthesis done right", USE, June1998.

[14] Medard Rieder, "A simple and complete Process for embedded Systems Development", Proceeding embedded world conference 2005, pages 876–885.