Roll your own Hardware Description Language

An Experiment in Hardware Development using Model Driven Software Tools

Guillaume Savaton
ESEO/TRAME
4 rue Merlet de la Boulaye, BP 30926
49009 Angers cedex 01 - France
guillaume.savaton@eseo.fr

Jérôme Delatour
ESEO/TRAME
4 rue Merlet de la Boulaye, BP 30926
49009 Angers cedex 01 - France
jerome.delatour@eseo.fr

Karl Courtel
ATMEL Nantes S.A
La Chantrerie - Route de Gachet,
B.P 70602,
44306 Nantes Cedex 3, France

ABSTRACT

The work presented here is a first study made at ESEO[1] by the research team TRAME (Model Transformations for Embedded Systems) for the semiconductor company ATMEL Nantes. This experiment consists in exploring the possibilities of Model-Driven Development (MDD) and model transformation in the context of digital hardware design.

Through the definition of several meta-models and model transformations dedicated to hardware design, some observations have been carried out regarding the use of an existing MDD tool (ATL suite of the ATLAS team of the LINA) in the context of industrial practices.

Keywords

Model-Driven Architecture (MDA™), Model-Driven Engineering (MDE), Model-Driven Development (MDD), Model-Integrated Computing (MIC), Domain-Specific Language (DSL), Hardware Description Language (HDL).

1. INTRODUCTION

Throughout the history of digital hardware design, engineers have been used to manipulating various representations for digital integrated circuits [Gajski83]. Hardware design languages and tools have been introduced to follow the evolution of circuit complexity and the needs for higher abstraction levels for hardware representation. As a result, design flows rely on tool chains that start from a high abstraction level – based on finite-state machines and data flow descriptions –, and end at the layout level, where the circuit is fully characterized in terms of geometry and electrical properties.

The design process can be seen as a set of refinement operations performed step by step (figure 1), each step adding implementation details to the initial model. With a variety of possible hardware representations and tool vendors, each tool in the chain relies on a different, often proprietary, model repository. Since these repositories were developed by different people at different times – following a bottom-up evolution in abstraction –, ensuring consistency and traceability between decisions taken at each step of the design flow is becoming more and more difficult [Mallis03].

Pushed by the fast increase of system complexity and the availability of more and more powerful design automation tools, designers have adopted hardware description languages (HDLs) such as VHDL [IEEE_1076_00] as the starting point for semi-automatic hardware generation and simulation. Today, integrating an existing hardware component in a new system can be done by importing and « compiling » its HDL description in a similar way as software designers do. The notion of virtual component – or so-called Intellectual Property (IP) core – has been defined in order to reflect this move towards a design methodology in which components are no longer concrete parts assembled on a printed-circuit board, but models that can be composed together to build a complete system on a single chip (SoC for System-on-Chip) [Keating99].

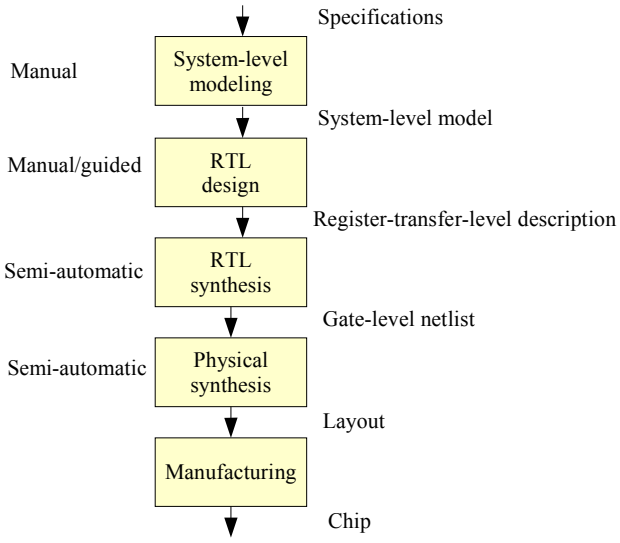

Figure 1. A typical hardware design flow

Since a complete system model can mix component models from various sources and various abstraction levels, there are multiple entry points in the system design flow. As a consequence, making all these models work together while meeting system requirements can become a real challenge.

2. COMPONENT REUSE IN HARDWARE DESIGN AT ATMEL: FROM A DOCUMENT-CENTRIC TO A MODEL-CENTRIC APPROACH

At ATMEL Nantes, a semiconductor company, it was observed that designers wishing to reuse code produced in earlier projects take more time in reverse engineering than in implementing new functions. In order to reverse this trend, ATMEL Nantes developed its own methodology, based on the literate programming approach [Knuth92].

The proposed documentation-centric approach obliges the designers to describe a hardware block in three ways (figure 2):

[1] A French high school of engineering

(1) literate, in natural language; (2) formal or semi-formal by using state-of-the-art models commonly used within the domain (truth tables, logic equations, state-transition diagrams, component interconnect diagrams, waveforms, etc); (3) code using an HDL. The strengths of this approach are: better knowledge management permitting exchange of good practices between engineers; a faster learning curve and less time spent in reverse engineering due to the use of widespread, easy-to-read models; better maintainability and reusability of the produced modules; more efficient code inspection.

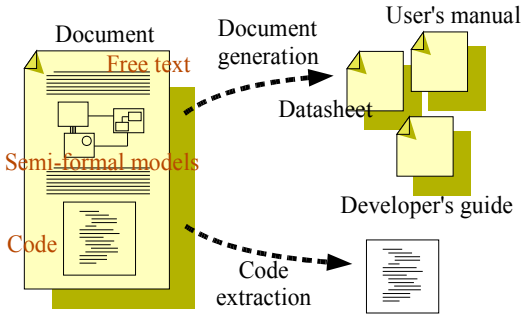

Figure 2. The literate programming approach

In spite of these improvements, the proposed design flow still suffers from limitations: Literate programming does not ensure consistency between the models used in the documents and the corresponding HDL code; Formal models and HDL code being equivalent representations of the same hardware, forcing the designer to write both appears as redundant work.

As a result, ATMEL would like to go one step beyond, towards a model-centric design methodology in which hardware refinement and code generation will be supported by automation and guidance tools.

3. MDD FOR HARDWARE DESIGN: A FIRST EXPERIMENT

As a first answer to ATMEL Nantes concerning the feasibility of such a methodology using existing tools, the TRAME team of the ESEO has proposed an MDD-based approach. A study has been engaged, in order to assess the interests of MDD approach in hardware development. Therefore, we focused our survey on studying the concepts of MDD and their applicabilities in an existing industrial development process rather than evaluating MDD tools. For this reason, we chose to work on a concrete and well-known transformation : the HDL code generation. ATMEL could then compare it with existing HDL code generators.

As our industrial partner required the use of free and open MDD tools, we used the model-driven software tool suite of the ATLAS team.

We followed the principles of Model-Integrated Computing (MIC) [Sprinkle04] by developing the following aspects: (1) capturing a subset of the hardware representations used by ATMEL Nantes (state-of-the-art models and HDLs) in the form of MOF meta-models; (2) writing transformations from ATMEL models to HDLs; (3) defining injectors and extractors for hardware models, so that they can be exploited in the other tools used by ATMEL (VHDL/Verilog simulators, synthesis tools, etc).

3.1 Meta-models

Two meta-models have been defined. The first one is called "Logic" and provides the basic elements for describing digital circuits using state-of-the-art models. It is the ATMEL DSL (Domain Specific Language). The second one, called "VHDL", describes the abstract syntax of the VHDL language. For the sake of simplicity in the context of the experiment, these meta-models are limited to a minimal subset.

The Logic meta-model

The *Logic* meta-model provides a notion of *component*, that allows to model a hardware system as a set of interconnected blocks. A component communicates with its environment through *ports*. In the current version of the meta-model, a *port* can be used to exchange one bit of data at a time.

Two kinds of components are supported: a *simple component* describes the behavior of a circuit (the relationship between its inputs, internal state and outputs). Presently, the supported simple components in *Logic* are: *truth tables*, *sets of logic equations* and *state machines*. A *composite component* describes the internal structure of a circuit. A *composite component* is composed of *component instances* connected using *wires*. Composite components provide a mechanism for hierarchical decomposition of a system. Figure 3 shows the structural part of the *Logic* meta-model.

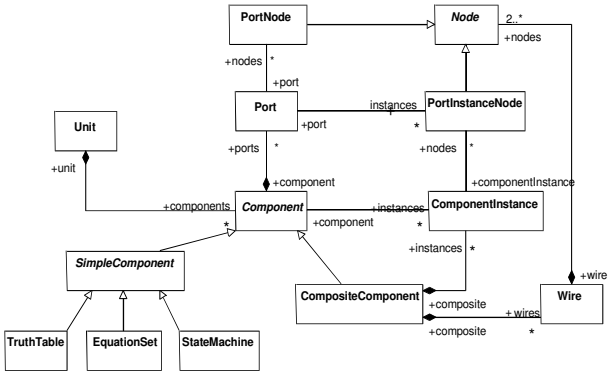

Figure 3. An fragment of the Logic meta-model

The VHDL meta-model

The VHDL meta-model reflects a subset of the abstract syntax of the VHDL language (Figure 4 depicts a subset of the structuration aspect).

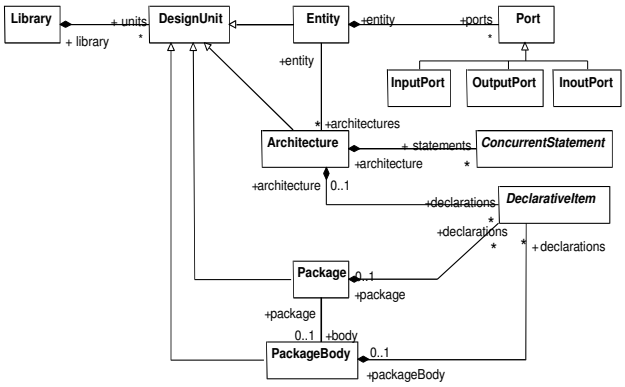

Figure 4. An fragment of the VHDL meta-model

In VHDL, the digital circuit is separated into an *entity* and one or several *architectures*. Basically, an entity models the interface of a circuit in terms of *ports* while an *architecture* models one possible implementation of the same circuit, in terms of *concurrent statements*.

Concurrent statements include: assignment statements; component instantiation statements; process statements, etc. A process statement itself describes a piece of behavior as a sequence of imperative constructs.

3.2 Transformations

Transformations that can be applied to hardware models include: automatic generation of document parts, transformation of hardware models to produce different representations of the same circuit (e.g. convert a state-transition diagram into an equivalent truth table); model analysis, validation, verification, extraction of metrics...

As previously stated in this paper, we focus on transformations for HDL code generation. Capturing state-of-the-art industrial practices was a strong requirement from ATMEL: for instance, a modeling environment for hardware design should allow fine-tuning the HDL output in terms of "writing style" and coding conventions.

The ATL tool suite

In order to automate the HDL code generation, we used the ADT (Atlas Development Toolkit) tool suite [Allilaire04], based on the Eclipse editor. It provides a complete environment for developing, testing and using the ATL (ATLas) transformation language [Bezivin03].

ATL is a hybrid language (a mix of declarative and imperative constructions) designed to express model transformations as required by the MDA™ to answer the QVT RFP [QVTRFP] issued by OMG. It is described by an abstract syntax (a MOF meta-model), a textual concrete syntax and an additional graphical notation allowing modelers to represent partial views of transformation models. A transformation model in ATL is expressed as a set of transformation rules.

Example of an ATL transformation rule

In the following examples, we show how a *port* in *Logic* is transformed into a *port* in *VHDL*.

Actually this transformation is not as straightforward as it may seem: as shown in figures 3 and 4, *ports* in VHDL are explicitly declared with a direction (*in*, *out* or *inout*) while the direction of a *port* in *Logic* is implicit. A transformation rule is defined for each kind of VHDL port. Below is the ATL code that transforms an input port from *Logic* to *VHDL*.

```
rule LogicPort2VHDLInputPort {
 from lgcPort : Logic!Port
   (port.isUsedAsInput() and not port.isUsedAsOutput() )
 to vhdlPort : VHDL!InputPort (
   name <- 'port_' + port.name,)
}
```

The difficulty of determining the direction of a port is hidden in the functions called *isUsedAsInput()* and is*UsedAsOutput()*.

In *simple components*, it is generally easy to identify input and output ports. For instance, a truth table is split into a set of *input columns* (class *InputColumn* of the meta-model) and *output columns* (class *OutputColumn*), each column being associated to a port of the component. In the case of a *composite component*, a *port* will be identified as an input (resp. an output) if it is connected to an input *port* (resp. an output *port*) of one of the contained *component instances*. As a result, there is a need for hierarchical – and recursive – exploration of a model.

We first attach a method *getBindings* to class *Port* of the *Logic* meta-model. Applied to a *port* of a *composite component*, this method navigates along the *wires* connected to this port and returns the set of *ports* of the inner *component instances* that are connected to it. In ATL, this method is written using the *helper* keyword followed by an OCL method definition:

```
helper context Logic!Port def: getBindings() : Sequence(Logic!Port) =
 if self.nodes->isEmpty() then Sequence {}
 else self.nodes->collect(n |
       n.wire.nodes->select(f| f.oclIsKindOf(Logic!PortInstanceNode))
                   ->collect(g| g.port)->flatten())->flatten()
 endif;
```

To determine *port* directions, we define methods *isUsedAsInput* and *isUsedAsOutput* in class *Port*:

```
helper context Logic!Port def: isUsedAsInput() : Boolean =
 if not self.column->isEmpty() then
  -- Port belongs to an input column of a truth table ?
  self.column->exists(c| c.oclIsKindOf(Logic!InputColumn))
 else
  -- Port is used in an expression ?
  not self.expressions->isEmpty()
  -- Port is bound to an input port of another component ?
  or self.getBindings()->exists(b| b.isUsedAsInput())
 endif;

helper context Logic!Port def: isUsedAsOutput() : Boolean =
 if not self.column->isEmpty() then
  -- Port belongs to an output column of a truth table
  self.column->exists(c| c.oclIsKindOf(Logic!OutputColumn))
 else
  -- Port is the target of an equation or an action of a state machine
  not self.assignments->isEmpty()
  -- Port is bound to an input port of another component
  or self.getBindings()->exists(b| b.isUsedAsOutput())
 endif;
```

3.3 Injectors and extractors

Logic is supposed to act as a front-end for standard HDLs. As such, there is a need for tools that could help capture models in a graphical way, and generate textual HDL code.

Designing a complete graphical environment for model creation was out of the scope of this experiment. However, to prevent the user from messing with *XMI* code, we proposed a simple textual syntax for *Logic* and generated a model *injector* from it. Basically, an injector is a parser that builds a *MOF* model from a textual representation. Below is an example of a truth table – the described component is a half adder – written in our textual concrete syntax for *Logic*.

```
table HalfAdder (a, b, c, s)
{
  in  a = ( L, L, H, H ),
  in  b = ( L, H, L, H ),
  out s = ( L, H, H, L ),
  out c = ( L, L, L, H )
}
```

The *table* keyword introduces a truth table component. It is followed by the name of the component and the list of its ports. Between curly brackets we place the *columns* of the tables: an input (resp. output) *column* is introduced by the keyword *in* (resp. *out*), followed by the port associated with the column and the list of possible values. *L* and *H* respectively stand for *low* and *high*.

Similarly, to complete the code generation flow, we produced a model *extractor* for *VHDL*. Basically, an extractor definition associates a textual template to each class of the metamodel. Here is the *VHDL* code generated from the half adder model:

```
entity HalfAdder is
  port( port_a, port_b : in bit; port_s, port_c : out bit );
end HalfAdder;
architecture Behavioral of HalfAdder is
begin
  process(port_a,port_b)
  begin
    if port_a = '0' then
      if port_b = '0' then
        port_s <= '0'; port_c <= '0';
      else
        port_s <= '1'; port_c <= '0';
      end if;
  else
    if port_b = '0' then
      port_s <= '1'; port_c <= '0';
    else
      port_s <= '1'; port_c <= '1';
    end if;
  end if;
  end process;
end Behavioral;
```

4. CONCLUSIONS

The study demonstrates the relevance of model transformation in the context of electronic design automation. With a growing variety of design languages and abstractions used in a hardware design flow, there is a real need for techniques to ease the development of new tools and to provide language-neutral environments.

While most MDA-based tools such as ATL are still in their infancy, our meta-models and transformation rules appeared as an interesting case study to evaluate the strengths and weaknesses of model transformation techniques. They revealed a need for a methodology to assist the designer in writing new transformations. Throughout the experiment, we could highlight several issues and identify a set of necessary features that transformation tools should include.

Among the several benefits expected from MDD and assessed by this study, two are extremely interesting for our industrial partner :

- The fact that hardware designers keep using their domain-specific representations;
- The ability to integrate domain-specific design practices and express them in the form of automated transformation rules. In fact, in the domain of HDL-based hardware design, there are recommended "writing styles" that can lead to more effective implementations in terms of speed or gate count.

However, in a context where the main issue is system complexity, the effectiveness of the approach and of the tools remains to be tested against a real-world case study (e.g. generating the VHDL for a multi-million gate system).

5. ACKNOWLEDGEMENTS

We would like to thank Frederic Jouault and Jean Bezivin of the ATLAS Team for their constant help and assistance. Parts of this work could not have been done without their advice and support.

6. REFERENCES

[Allilaire04] F. Allilaire, T. Idrissi. "ADT, Eclipse Development tools for ATL", Internal report, 2004 http://www.sciences.univ-nantes.fr/lina/atl/publications/ADT_AllilaireIdrissi.pdf

[Bezivin03] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, J. Rougui. "First Experiments with the ATL Transformation Language: transforming XSLT into Xquery", 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture, Anaheim, California, 2003.

[Gajski83] D. D. Gajski and R. H. Kuhn. "Guest Editors Introduction", New VLSI Tools, IEEE Computer, vol 16, 2, pp 11--14, 1983

[IEEE_1076_00] IEEE/DASC/VASG. "Draft IEEE Standard VHDL Language Reference Manual", IEEE P1076.2000/D3, 2000

[Keating99] M. Keating and P. Bricaud. "Reuse Methodology Manual for System-on-a-Chip Design", Kluwer Academic Publishers, 1999

[Knuth92] D. Knuth. "Literate Programming", CSLI, ISBN 0-937073-80-6,1992

[Mallis03] David Mallis and Don Cottrell. "OpenAccess: The Standard API for Rapid EDA Tool Integration", Silicon Integration Initiative Inc., 2003

[QVTRFP] Object Management Group : OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP

[Sprinkle04] Sprinkle J., "Model-Integrated Computing", IEEE Potentials, Vol. 23, No. 1, pp. 28-30, February, 2004.