An Experiment in Using Model Driven Development:
Compiling UML State Diagrams into VHDL

David Akehurst[1], Gareth Howells[1], Klaus McDonald-Maier[2], Behzad Bordbar[3]

[1]University of Kent, Canterbury, UK
{D.H.Akehurst, W.G.J.Howells}@kent.ac.uk
[2]University of Essex, Colchester, UK
kdm@essex.ac.uk
[3]University of Birmingham, Birmingham, UK
B.Bordbar@cs.bham.ac.uk

Software systems are becoming increasingly more complex. Model Driven Development (MDD) techniques are being proposed as a potential means to manage that complexity. In this paper, we discuss the results of experimenting with the use of Model Driven Development techniques as a means of compiling UML State Diagrams into synthesisable VHDL code. These results show that although MDD provides a potentially useful technique, there are aspects where we still need significant improvement in the MDD tools and techniques in order to make this an easily useable approach to the task.

1 Introduction

As our software systems grow ever more complex, we require tools and techniques to aid the developer in managing that complexity. A commonly used technique is to raise the level of the language in which the system can be developed and recently there has been a move towards more and more use of graphical and domain specific languages. In particular Model Driven Development techniques (MDD [30]) are being proposed as a successful way to better manage the increasing complexity.

The ModEasy [20] project explores provision of automated support for mapping embedded system designs on to both low level implementations and system verification toolkits. There is a clear advantage in providing this kind of automated support – namely that there is a single specification written by the designer, which is used to both verify and implement the system, thus avoiding the significant problem of the introduction of differences between the verified and implemented versions of the system.

Part of the work of the ModEasy project requires the development of a framework for deriving VHDL specifications from UML state diagrams (one of the UML mechanisms for describing the behaviour of a component) and the definition of a set of rules that enable automated procedures to generate VHDL code from UML notations.

The research on which this paper is based is twofold. Firstly, an investigation into the use of UML State Machines for specifying the behaviour of embedded system components. The results of this aspect of the research are reported in other publications.

This paper focuses on the experience gained from the use of MDD techniques as a way to create a State Machine to VHDL compiler. The paper treats the State Machine to VHDL problem as a means to exercise the MDD techniques, and highlights interesting and problematic aspects of that exercise.

Section 2 of the paper gives a brief overview of MDD, State Diagrams and VHDL for those readers for whom they are unfamiliar. Section 3 contains an overview of the case study. Section 4 discusses our experiences of using MDD techniques on this case study. Sections 5 and 6 discuss related work and conclude the paper.

2 Background

Model Driven Development

Model Driven Engineering (MDE) or Model Driven Development (MDD) is an approach to software development in which the focuses is on Models as the primary artefacts in the development process, with Transformations as the primary operation on models, used to map information from one model to another.

In general, we can view Model Driven Development as a general principle for software engineering that can be realised in a number of different ways (using different standards) and supported by a variety of tools. One of the most common realisations of MDD is using the Object Management Group's (OMG) Model Driven Architecture (MDA) [23] set of standards.

Within MDA the central idea of object composition is replaced by the notion of model transformation. The idea of software systems being composed of interconnected objects is not in opposition with the idea of the software life cycle being viewed as a chain of model transformations; rather they are seen as complimentary techniques, with MDA build on top of OO.

MDD is an evolving paradigm with many expectations on its final potential and with much research and development needed before it will meet those expectations. A comprehensive review of MDD techniques is given (amongst other places) in [7].

UML State Diagrams

The UML state diagram formalism is a variant of Statecharts invented by David Harel [12]. Harel introduced diagrams that extend traditional state-transition diagrams with the notions of hierarchy and concurrency. A state diagram is a technique to describe the behaviour of a system and represents the event triggered flow of control where objects change state by means of transitions. There are a number of books and papers about UML state diagrams, although the definitive definition is given as part of the UML standard [25].

VHDL

Very High Speed Integrated Circuits Hardware Description Language (VHDL) is a hardware description language used to describe the structure and the behaviour

of digital electronic systems [16]. VHDL allows three styles of design:

1. The *Structural* approach describes how the system is decomposed into sub-systems and how those sub-systems are interconnected.
2. The *Behavioural* approach which permits description of the behaviour of a component by means of high-level language constructs, similar to other high level programming languages.
3. The *Dataflow* approach describes the circuit in terms of how data moves through the system from input to output.

VHDL specifications of complete systems typically contain a number of mixed-type designs. The predominant feature of VHDL is the ability to simulate the design before committing it to manufacturing, allowing designers to quickly compare alternatives and verify the behaviour of the circuit.

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 circuit, in terms of concurrent and sequential statements.

VHDL was originally designed as a language for simulating digital circuits, not as a language from which to directly generate a hardware implementation. Consequently, many VHDL constructs are not supported by synthesis software. The set of concepts from which it is possible to generate hardware are a subset of the full VHDL language, known as Register Transfer Level (RTL). The syntax and semantics that can be used in common by all compliant RTL synthesis tools to achieve uniformity of results have been defined in [15]. It defines the subset of [16] that is suitable for RTL synthesis as well as the semantics of that subset for the synthesis domain.

3 Overview of the Case Study

The problem we wish to solve using MDD techniques is illustrated in Figure 1. The aim is to enable us to compile a UML based specification of an embedded system into VHDL code – which can be subsequently synthesised for implementation onto an FPGA.

Ideally, the UML specification should consist of a mixture of Component and State Diagrams. The Component diagrams will define a configuration of predefined modules and their connectivity. State diagrams facilitate the specification of a state based behaviour for any component whose behaviour can be thus expressed; some components will require other mechanisms to define their non-state based behaviour. However, that issue is out of scope for the purpose of the example in this paper.

The overall problem is broken down into a number of tasks to which we apply different aspects of MDD technology. We address these tasks using different MDD techniques as follows:

State Diagram Editor: A State Diagram editor is essentially an editor for a Domain Specific Graphical Language. Typical graphical language tool/editing support should be provided enabling a pleasant user experience for entering State Diagram specifications. There are two options to providing a solution to this aspect: Either by using an existing UML editing tool and

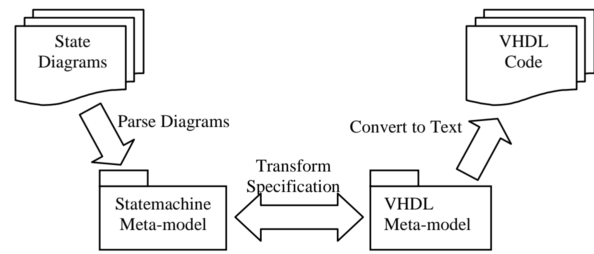

Figure 1

employ a model interchange format (e.g. XMI) to import the specification; or to build a bespoke (DSL). We currently use the second of these options, although we plan to enable import from XMI in the future.

State Diagram and VHDL Meta-Models: The MDD approach to manipulating languages implies that one should use an abstract representation (model) of the language commonly referred to as a *meta-model*. To specify the meta-model, a set of class diagrams must be developed that capture the abstract concepts of the language and the relationships between them. To subsequently provide tool support for a language model, these class diagrams are used as the basis to provide a repository for expressions or specifications in that language; these are often referred to as “instances” of the language model. The UML State Diagram meta-model is defined for us in the standard [25], but there is no standard VHDL meta-model.

State Diagram to VHDL Transformation: Using the MDD approach, the primary compilation process is captured in a model transformation that maps concepts from the State Diagram meta-model onto concepts in the VHDL meta-model. Recently a variety of model transformations techniques and tools have been developed, many based on the OMG’s Queries Views and Transformations document (QVT) [28]. There are issues here regarding the specification and the execution (and/or implementation) of the transformation.

VHDL Code: The final stage in our compilation process is the creation of text files that represent the VHDL version of our specification. These text files, or VHDL code, are used as input to dedicated synthesis tools for generating the final data used to implement an FPGA. The MDD technique for creating text from a model is to make use of a model-to-text transformer; a common technique for supporting such a transformer is the use of a template language, and we use this technique.

4 Experiences

This section introduces our experience of using MDD techniques as a means to implement the separate parts of the tool chain which enables us to compile a UML State Diagram specification into VHDL Code. We look at five different aspects: implementation of the meta-models, transformation from DSL editor to meta-model; transformation between two meta-models; transforming a VHDL model into VHDL code; and executing or implementing a transformation.

4.1 Implementing the Meta-Models

In order to produce a model transformation that maps one language to another, it is necessary to define models of the two languages (i.e. meta-models).

There are two basic approaches to supporting the implementation of a meta-model:

1. to implement the models directly in Java (using an approach such as EMF [14]),
2. to implement the models as instances in a meta-model repository (using an approach such as MDR [18].

Both of these choices have their advantages and drawbacks. Use of a meta-model repository provides a quick way to provide support for a given model. One needs only to provide a specification of the model (possibly using XMI [26]) and the meta-model repository can be used as a repository for instances of the given model. Another advantage is that the meta-repository should easily provide support for serialising and un-serialising instances of the given model in a standard format (again potentially XMI). The drawbacks of this approach are efficiency issues introduced by the meta-level objects that represent each of the given model objects, and the consequent reliance on the (probably 3rd party) meta-model repository itself. This latter issue is partially resolved by the standardization of the Java Metadata Interface (JMI) [17] that forms part of the libraries issued with Java 5, which provides a standard interface for MOF repositories, i.e. JMI (version 1.0) defines a Java mapping for the MOF (version 1.4).

The direct implementation approach requires more effort up front (i.e. specifying the mappings from modelling concepts to programming concepts). Additionally, serialising model instances is more complex. The advantages of the direct approach are that there is no reliance on a third party library, and if you have control of the code generation templates you can have complete control of the implementation patterns and hence some control over the efficiency choices.

Our State Diagram meta-model is defined in accordance with the UML 2 set of standards and uses some of the more complex modelling concepts (such as subsetting and refinement of properties). In part due to efficiency reasons, but primarily because we do not wish to be reliant on third party libraries, we chose the first of these techniques as a means to support implementation of the meta-models.

Even though the modelling community has been using MOF and UML [24, 25] based languages for several years, and there are a number of tools that generate Java code from UML models, there are still some significant issues regarding how to implement some of the concepts (especially due to the introduction of some new ones with the advance to UML 2.0). In particular, one of the major abstractions used in object-oriented (OO) modelling languages such as MOF and UML is the concept of an association. This concept does not exist in OO programming languages such as Java (or any of the other mainstream OO languages). Consequently, in order to generate code from a UML or MOF model, there is a complex process involving the mapping of the modelling concepts on to the programming language concepts.

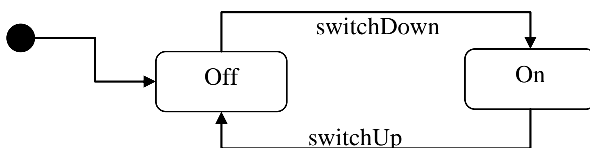

Figure 2

The major results of a review into which tools can be used to implement a MOF 2 meta model can be found in our submitted work [4]; to summarise, we found none at the time of writing that entirely met our requirements, thus motivating the primary work of that paper. We use techniques based on the work of that paper to generate a Java based implementation of the standard OMG meta-model specification for UML state diagrams. We created our own VHDL meta-model of the RTL subset.

4.2 Editor to Model Transformation

There is a significant quantity of research on the parsing of visual languages. However, as yet, no standard or commonly used process has emerged. Much of the work is in relation to graph grammars and graph transformations [6, 9], and some makes use of model transformation as a technique [1].

Our implementation makes use of a common Domain Specific Language (DSL) editor that has a generic model of diagrams; elements in the diagram model are mapped to different symbols in the visual representation. The editor makes available an instance of the generic diagram model that represented the expression drawn. Thus the transformation to language meta-model is required to be a definition of the mapping between the diagram meta-model and the target language meta-model.

4.3 Model-to-Model Transformer Specification

Using a QVT like language (KMTL [3]), we found the writing of the majority of the transformations to be quite straight forward. However, there is an issue, specific to the use of model transformation as a way to define a compiler, which we feel if addressed, would make the process simpler. The issue is illustrated below, but in general it is with respect to transformation rules that map to a target *pattern of objects* that can be simply expressed in a language designed for representing target model expressions; however, in the language for defining a model transformation, these patterns can be complex and error prone to construct.

Although we have used KMTL as our transformer specification language, the issue raised is applicable to most of the QVT based transformation languages.

An example of the issue

The top level mapping from a State Machine to VHDL requires us to construct an architecture body plus some additional code. The bulk of the complexity in this transformation comes from the need to setup this 'boiler plate' VHDL code that surrounds the code specific to a particular state machine. To illustrate this, the example state machine of Figure 2, should compile to the VHDL code shown in Table 1. The VHDL code in bold type is the 'boiler plate' behaviour code for the architecture that is common to the compilation of all state machines.

Our first attempt at specifying this is shown in Table 2. The statements of the architecture body are defined to be a pattern of literal objects that represent the 'boiler plate' code illustrated in Table 1. The state machine specific code is subsequently linked in via the two variables 'sas' and 'caseSt' (highlighted in bold type) and produced by other mapping rules.

```
ARCHITECTURE behviour OF entity IS
  TYPE topReg IS ('On','Off');
  SIGNAL cs : topReg
BEGIN
  PROCESS (clk) BEGIN
    IF clk='1' and clk'event THEN
      IF reset='1' THEN
        cs <= 'Off';
      ELSE
        CASE (cs) IS
         WHEN 'On' =>
          IF switchUp='1' THEN
           cs <= 'Off';
          ELSE
           cs <= 'On';
          END IF
         WHEN 'Off' =>
          IF switchDown='1' THEN
           cs <= 'On';
          ELSE
           cs <= 'Off';
          END IF
        END CASE
      END IF
    END IF
  END PROCESS
END ARCHITECTURE
```

Table 1

This is a correct specification; however, it is not what you would call 'readable' and it is also easy to make a mistake when writing it. The readability issue comes from the need to express, in the transformation language, a pattern of object literals. In this case, the pattern of objects we wish to represent is better expressed in its natural syntax, rather than in the syntax for expressing object literals.

A potential solution

To facilitate this, we suggest that a mechanism be introduced into rule specification languages, which allows us to specify object literals in an alternative syntax. An example of the use of such a mechanism is illustrated in Table 3.

Building a parser, interpreter or compiler for a transformation language that includes such a mechanism will certainly have its challenges; but we do not believe that it would be impossible.

4.4 Model-to-text Transformation

Within the domain of model transformations, in addition to the QVT [28], the OMG as issued a separate request for proposals specifically targeting the issue of converting models into text [27]. The predominant approach to this issue make use of template languages similar to Velocity [32] though rather than being based on Java, they utilise an object navigation languages such as OCL for navigating the MOF model.

We have made use of just such an OCL based template language [5] built on top of the Kent OCL library [22]. An example specification is shown in Table 4, which generates the code for a VHDL Case statement.

We have found the technique of a template language to be a very simple and effective mechanism for mapping our meta-model of VHDL into text files. However, there are a couple of significant drawbacks we have encountered using this approach.

```
rule StateMachine2ArchitectureBody

sm:StateMachine [region=regs]

ab:ArchitectureBody [
  name='behaviour',
  declarativeItem=decls
  statement = Set {
   Process{
     statement=OrderedSet{
       IfStatement{
         option=OrderedSet{
           IfOption{
             condition=Expression{
              value='clk=\'1\' AND
              clk\'event'
             }
             statement=OrderedSet{
              IfStatement{
                option=OrderedSet{
                  IfOption{
                    condition=Expression{
                      value='reset=\'1\''
                    }
                    statement=sas
                  }
                }
                else=caseSt
  }} } } } } }}
]

where {
  Region2SetDeclaration(regs,decls) and
  Region2InitialSigAssignment(regs,sas) and
  Region2CaseStatement(regs,caseSt)
}
```

Table 2

Firstly, there is a clash of interests between the layout of the template specification and the layout of code that is generated. A neatly laid out specification introduces excess white space into the output code, whereas enabling nicely laid out output code requires the template specification to be unreadable. Secondly, another issue related to formatting the output, there is no easy to use mechanism for producing nested indenting in the output within the template language used.

```
rule StateMachine2ArchitectureBody

sm:StateMachine [region=regs]

ab:ArchitectureBody [
  name='behaviour',
  declarativeItem=decls,
  statement = <template language='VHDL'>
                PROCESS (clk) BEGIN
                  IF clk='1' AND
                     clk'event THEN
                    IF reset='1' THEN
                      <%sas%>
                    ELSE
                      <%caseSt%>
                    END IF
                  END IF
                END PROCESS
              </template>
]

where {
  Region2SetDeclaration(regs,decls) and
  Region2InitialSigAssignment(regs,sas) and
  Region2CaseStatement(regs,caseSt)
}
```

Table 3

```
context
vhdl::SequentialStatement::CaseStatement
def : generate : String =
let
  alts = self.alternative,
  oths = self.default
 in
<template>
  CASE <self.expression.value> IS
    <foreach alt in alts
     let
       chc=alt.choice,
       stms = alt.statement
     in>
       WHEN <foreach lit in chc>
            <lit.as(EnumerationLiteral).value>
            </foreach> <'=>'>
          <foreach st in stms>
           <st.generate>
          </foreach>
    </foreach>
      <if oths->isEmpty then><else>
       WHEN OTHERS <'=>'>
        <foreach st in oths>
          <st.generate>
        </foreach>
      <endif>
  END CASE;
</template>
```

Table 4

These issues, related to formatting the output lead us to believe that an additional module is required in our compilation process, one that formats the output code, according to specifiable user preferences.

4.5 Implementing the Transformation

This transformation has been implemented (manually) using SiTra a Simple Transformation library [2]. This library consists of two interfaces and a class, which between them give support for implementing model transformations in Java. The two interfaces are as shown in Table 5; and the provided class is a simple implementation of the Transformer interface.

Initial attempts were made to implement this transformation using a full Model Transformation Framework, however, the developer on the project was new to the concepts of MDD and MTs and found that the additional overhead of the full (and complex) MTF to be daunting and difficult to use.

The use of this simple Java library instead, meant that the developer could stay in the familiar environment of Java programming, and yet gain some support for building an application using the concepts of model transformations and transformation rules. Further empirical studies are planned to test whether this is a common experience.

5 Related work

This section briefly discusses several other projects that have investigated transforming UML state diagrams to VHDL and also graphical entry tools that can generate VHDL from state machines diagrams. There are several studies and tools regarding VHDL code generation from system-level specifications [11, 13, 21, 31, 33], and there are a few which address generating VHDL from the Unified Modelling Language (UML) [8, 10, 29].

There are several differences between the other approaches and the investigation discussed in this paper, which focuses on using complementary MDA techniques and on development issues specific to electronic embedded systems design.

The researchers in [8] undertook an approach to first translate UML models to a language called SMDL (the language with formal semantics and high-level concepts such as states, queues and events), which is then compiled into VHDL code. They did not use MDA techniques. In our approach we try to map the high-level concepts as directly as possible to the target language concepts.

The authors of [19] developed a prototype system for generating the VHDL specifications from the UML models. Their generated VHDL code is for simulating and verifying the UML models and not for implementing it on the hardware. Their approach is similar to the one presented here in that they transformed models using homomorphic mappings between dissimilar structures while preserving meta-model class associations in a way that resembles an MDA technique.

The research team in [10] presented MODCO, a tool that transforms UML state diagrams directly into synthesisable VHDL using MDA technique. Their transformation approach is very similar. However, the authors targeted flat state-transition diagrams without covering hierarchy, concurrency and the notion of actions, only supporting a small subset of UML state diagram constructs.

There are several existing tools that can generate VHDL from state diagrams [13, 31]. Nevertheless, it appears to be a limited number of tools and techniques that can generate VHDL from UML state diagram specifications [33]. However, none of them are built using MDD techniques, and none support the full range of concepts in UML state diagrams.

6 Conclusion

The aim of this paper has been to investigate the use of Model Driven Development techniques as a means to develop a compiler from the high level graphical language of UML State Machines into the lower level textual language of Synthesisable VHDL. To facilitate this we have: made use of and implemented meta-models of the two languages; defined and implemented two model transformations, one from the model of a diagram into a model of state diagrams, and another that maps the state diagram onto concepts in the VHDL language; and written a model-to-text transformation to map VHDL models into VHDL code.

```
interface Rule<S,T> {
  boolean check(S source);
  T build(S source, Transformer t);
  void setProperties(T target, S source,
                     Transformer t) ;
}
interface Transformer {
  Object transform(Object source);
  List<Object> transformAll(
              List<Object> sourceObjects);
  <S,T> T transform(
             Class<Rule<S,T>> ruleType,
             S source);
  <S,T> List<T> transformAll(
                 Class<Rule<S,T>> ruleType,
                 List<S> source);
}
```

Table 5

We have demonstrated the potential for MDD techniques as a method for producing such a compiler. The two languages are significantly different in structure for the issue to be non trivial, and we feel that the technique of model transformation has proved, potentially, to be a good mechanism for specifying the mapping from one to the other. However, there are still a few issues to address before the process is truly effective and easy to use.

We identify three specific areas where we feel improvements are necessary:

1. The need for a mechanism within a transformation specification language that makes use of a concrete syntax of the source and/or target models. We have illustrated a potential method of doing this for text based languages, but more investigation is required in this area.
2. Simple mechanisms for defining and implementing model transformations that are accessible to programmers and do not require significant overhead in terms of time or supporting framework. We have developed SiTra [2] as 'a' solution to this issue and intend to carry out some empirical trials to test if it is generally effective.
3. Associated to the notion of a model-to-text transformation, it is necessary to provide a mechanism for formatting the output of such a transformation, that is separate to the transformation specification. Such a mechanism is unnecessary if the output of a compiler is not going to be read by a human; however, when mapping high-level graphical languages onto current programming languages (VHDL in our case) the output is required to be human readable.

References

1. Akehurst, D.H.: An OO Visual Language Definition Approach Supporting Multiple Views. VL2000, IEEE Symposium on Visual Languages (September 2000)
2. Akehurst, D.H., Bordbar, B., Evans, M., Howells, W.G., McDonald-Maier, K.D.: SiTra: Simple Transformations in Java. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences), Vol. 4199. LNCS, Genova, Italy (October 2006) 351-364
3. Akehurst, D.H., Howells, W.G., McDonald-Maier, K.D.: Kent Model Transformation Language. Model Transformations in Practice Workshop, part of MoDELS 2005, Montego Bay,Jamaica (October 2005)
4. Akehurst, D.H., Howells, W.G., McDonald-Maier, K.D.: Implementing Associations: UML 2.0 to Java 5. Journal on Software and Systems Modeling (to appear)
5. Akehurst, D.H., Patrascoiu, O.: Tooling Metamodels with Patterns and OCL. In: Evans, A., Sammut, P., Willans, J.S. (eds.): Metamodelling for MDA: First International Workshop, York, UK (November 2003) 203
6. Bardohl, R., Taentzer, G., Minas, M., Schurr, A.: Application of Graph Transformations to Visual Languages: Handbook on Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools. World Scientific (1999)
7. Berre, A., Hahn, A., Akehurst, D.H., Bezivin, J., Tsalgatidou, A., Vermaut, F., Kutvonen, L., Linington, P.F.: State-of-the art for Interoperability architecture approaches. InterOP Network of Excellence - Contract no.: IST-508 011, Deliverable D9.1 (November 2004)
8. Bjorklund, D., Lilius, J.: From UML Behavioural Descriptions to Efficient Synthesizable VHDL. Proceedings of 20th IEEE Norchip Conference, Copenhagen, Denmark (November 2002)
9. Costagliola, G., Tortora, G., Orefice, S., DeLucia, A.: Automatic generation of visual programing environments. IEEE Computer **28** (March 1995) 56-66
10. Coyle, F., Thornton, M.: From UML to HDL: a Model Driven Architectural Approach to Hardware-Software Co-Design. Information Systems: New Generations Conference (ISNG), Las Vegas NV, USA (April 2005) 83
11. Daveau, J.-M., Marchioro, G.F., Valderrama, C.A., Jerraya, A.A.: VHDL generation from SDL specification. Hardware Description Languages and their Applications (CHDL'97), Toledo, Spain (1997)
12. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8** (1987) 231-274
13. HDL-Designer. http://www.mentor.com
14. IBM: Eclipse Modeling Framework. http://www.eclipse.org/emf/
15. IEEE: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis. IEEE, IEEE Std 1076.6 (1999)
16. IEEE: IEEE Standard VHDL Language Reference Manual. IEEE, IEEE Std 1076 (2000)
17. Java Community Process: Java Metadata Interface (JMI) Specification. http://java.sun.com/products/jmi/
18. Matula, M.: NetBeans Metadata Repository. http://mdr.netbeans.org
19. McUmber, W.E., Cheng, B.H.: UML-based Analysis of Embedded Systems Using a Mapping to VHDL. IEEE High Assurance Software Engineering (HASE 99), Washington, DC, USA (November 1999)
20. ModEasy-Team: ModEasy Project. http://www.lifl.fr/modeasy/
21. Narayan, S., Vahid, F., Gajski, D.D.: Translating system specifications to VHDL. IEEE European Design Automation Conference, Amsterdam, netherlands (1991)
22. OCL-team: Kent OCL library. www.cs.kent.ac.uk/projects/ocl
23. OMG: Model Driven Architecture (MDA). Object Management Group, ormsc/2001-07-01 (July 2001)
24. OMG: Meta Object Facility (MOF) 2.0 Core Specification. Object Management Group, ptc/03-10-04 (October 2003)
25. OMG: UML 2.0 Superstructure Specification. Object Management Group, ptc/03-08-02 (August 2003)
26. OMG: XML Metadata Interchange (XMI), v2.0. Object Management Group, formal/03-05-02 (May 2003)
27. OMG: MOF Model to Text Transformation Language RFP. Object Management Group, ad/2004-04-07 (April 2004)
28. OMG: MOF QVT Final Adopted Specification. Object Management Group, pct/05-11-01 (November 2005)
29. Savaton, G., Delatour, J., Courtel, K.: Roll your own Hardware description Language: An Experiment in Hardware Development using Model Driven Software Tools. Best Practices for Model Driven Software Development, OPPSLA & GPCE Workshop, Portland, Oregon (October 2004)
30. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software **20** (Sept 2003) 19-25
31. StateCAD. http://www.xilinx.com
32. Velocity: Velocity Template Language. http://jakarta.apache.org/velocity/
33. WithClass. http://www.microgold.com