Object-Oriented High-Level Modeling of System Components for the Generation of VHDL Code

Karlheinz Agsteiner Dieter Monjau Sören Schulze

Department of Computer Science
Technical University of Chemnitz-Zwickau
Chemnitz, Germany
e-mail: kag@informatik.tu-chemnitz.de

Abstract
This paper describes a method for the design of digital systems that automatically derives an executable prototype of a system from an informal specification. Our method bases on an object-oriented hierarchy of classes which describe the functions of components. A tree of instances of these classes representing a certain system is built depending on the specification. VHDL descriptions of all classes involved in this tree are transformed into VHDL code for the complete system. Our approach emphasizes a bottom-up procedure and reuse of existing components.

1 Introduction
Several different approaches towards the description of hardware at system level have been suggested [7], for example SpecCharts [13], Statecharts [5], speedCHARTS [11], SDL [4], Extended Sequence Charts [5]. Important models used as a basis by these approaches are communicating concurrent processes, extended finite state machines, and hierarchical descriptions. The methods for system level design with these concepts usually follow a top-down approach: Starting with an initial graphical representation, a sequence of refinement steps is executed. The intermediate representations are attributed by VHDL annotations. This refinement process lasts until a complete description of the system’s behavior is built that can be compiled into VHDL code which in turn serves as a foundation for a simulation or synthesis.

Our method of designing digital systems uses a bottom-up approach based on an object-oriented domain model. For every concept met in a certain domain this model includes a description of all variants of the type. In case of functional units this includes all information concerning its behavior as well as quantitative and qualitative properties. This includes the unit’s interface, descriptions of its behavior using different approaches (e.g. VHDL, Petri-nets, temporal logic), hierarchical structuring, specialization, attributes, and generic parameters. In order to build a complete hierarchy from this set of types, appropriate relations between them have to be defined. In our case, two relations were needed:

- an is-a-Relation forming a taxonomical hierarchy which allows to represent specialization and
- a has-parts-Relation building a decompositional hierarchy.

CLINT (class and instance notation), a notation defining the notions class and instance is presented providing a textual counterpart to the graphical representation.

Digital systems are described by an informal specification. A knowledge based configuration system transforms this specification into an instance tree. The VHDL description of a system with specific properties is automatically generated from an instance tree using a model generator that works bottom-up: Starting with the behavior of leaf components which don’t have any subcomponents, the behavior of increasingly complex components is generated as a function of the behaviors of the subcomponents it consists of until the whole system is treated. The resulting specification of the system’s behavior uses plain VHDL and is suitable for simulation or synthesis if the behavior description of every class is.

This paper is divided into four major parts: First a description of the components our system consist of and the representations they manipulate gives a brief summary of our project. Then we introduce our object-oriented model by describing the notions involved and comparing the model to classical object-oriented approaches. The following section concerns the syntax of CLINT and an algorithm to transform CLINT descriptions into VHDL. Finally, we show a rough sketch of our model generator algorithm.

2 Representations and Transformations in our approach

Fig. 2 shows the representations involved in our project and the transformations between them as they appear in the design cycle. The figure is divided into five regions:

- A specification level contains knowledge about how to specify a certain system. This knowledge is partitioned into two parts: An object-oriented class hierarchy represents concepts met
Figure 1: Representations and transformations.

when specifying systems and relations between these classes, a constraint net captures dependencies that cannot be modeled by this hierarchy. Typical specification classes in the domain of RISC processors are "what kinds of load/store instructions does the CPU offer" or "is the CPU superpipelined".

Using a specification editor the designer creates a specification based on the hierarchy. This specification is merely a tree of class instances connected properly and a set of constraints concerning requirements the system should fulfill. A complete specification is mapped onto a partial tree of instances in the function level.

- The function level contains the core of our system: A set of classes that represent functions a system or a part of it can fulfill. Again, this level consists of a class hierarchy with constraint net. An instance tree at this level is called construction. It shows how a complete system is decomposed into the functions it fulfills which are again decomposed into sub-functions, and so on. Typical classes on this level are "a register file" or "an addition function".

The kernel of our approach, a knowledge based configuration system expands and connects the partial construction corresponding to the function hierarchy until a complete construction is found. Knowledge directs the configuration process whenever there are different options like selecting one of a set of specializations that can fulfill a certain function. Afterwards, a hierarchy and an instance tree can be transformed into a set of CLINT declarations.

- These CLINT declarations serve as the basis for a model generator. First, the CLINT class descriptions are transformed into VHDL entity declarations and special architecture bodies. Then, these VHDL declarations and a set of instance definitions build the input to the model generator. The model generator produces a set of VHDL files.

- The result of the model generator is prototype of the system constructed before in form of a set of VHDL entity declarations (one for each instance in the construction) and associated behavior-architecture bodies. They can be fed into tools like Synopsys [12] for simulation or synthesis purposes.

In this paper we ignore the specification level and confine ourselves to the object oriented model transformations from CLINT to VHDL, and model generation. For a complete descriptions of all levels present in our model cf. [2].

3 An object-oriented class hierarchy

A class hierarchy evolves in a natural way when a set of systems belonging to the same domain is decomposed. The exact appearance of this hierarchy, i.e. what kinds of relations should describe connections between classes, if there should be several disjunct “categories” of classes such as abstract functions vs. concrete hardware/software units implementing a set of functions and so on, depends on the domain
modeled by the classes. A reasonable class hierarchy should not be too restricted in order to permit a smooth integration of every boundary between two classes into the hierarchy. On the other side, if the hierarchy gets too complex dependencies between two classes can be realized in many different ways which leads to unnatural and inconsistent class systems.

The object-oriented paradigm forms an appropriate framework for a hierarchy. In fact, introducing the concepts of inheritance and decomposition well known in object-oriented software design as relations in our class hierarchy yields a class system that can easily represent most aspects of typical domains.

### 3.1 The conceptual hierarchy

A conceptual hierarchy is defined by a finite set of classes, a specialization relation is-a, and a decomposition relation has-parts. Both relations are acyclic, every class has at most one is-a-parent. has-parts has an arity and an optional task identifier associated to it. Furthermore, the whole hierarchy has exactly one root-class in is-a^{-1}∪has-parts. The formal definition of our class hierarchy is given in [1].

Every class is enriched by a set of attributes describing qualitative and quantitative properties of an instance of that class (i.e. generic parameters) and a set of behavior descriptions called views. These represent different aspects of an instance's behavior. Providing different views allows not only to simulate or synthesize a system using VHDL but also to check the unit for bottlenecks using Petri-nets ([8]).

Fig. 3.1 shows the graphical representation of an example hierarchy. In this figure, classes are depicted as nodes, is-a connections are shown as arrows from the “child” class to the “parent” class, has-parts is portrayed by arcs crossed by a circle segment. The boxes at the right part of the figure show attributes, ports, and views of class REGB_1, a register file with REGBANKS non-overlapping banks, generic size (REGFILESIZE) and width (REGWIDTH). Since REGB_1 is only a specialization of the abstract class REG it doesn’t define any new ports or attributes. The VHDL view contains a behavior-architecture-body implementing the behavior of an instance of this class based on n =REGFILESIZE units of class REGFD.

### 3.2 Constructions

As noted above, all data about one certain system that can be modeled by the class hierarchy are represented as an instance tree called construction. This tree corresponds to a complete system if has the following properties (Again, for a formal definition cf. [1]):

- The root element of the tree is an instance of the root class or a specialization of it.
- If an instance is in the tree then it has as many parts as described by the class hierarchy. These parts are instances of the right class or of specializations of it.

Every object in an instance tree possesses a set of attributes. This set is always a superset of the attributes defined by the object’s class.

![Figure 2: An example hierarchy describing register files](image)

### 3.3 The conceptual hierarchy as an object-oriented model

The concepts defined above match very closely to notions introduced by object-oriented modeling. For example, [9] defines the notions:

- **class diagram** as
  
  ...a schema, pattern, or template for describing many possible instances of data. A class diagram describes object classes.[9, p. 23]

- and an **instance diagram** as a way to express relations between instances.

These terms correspond to our notions conceptual hierarchy and instance tree, respectively. A closer examination of our representation shows that nearly all of the typical features of an object-oriented model like the one defined in [9] are present:

- **attributes**: Like mentioned in 3.1 and 3.2, attributes are an important constituent of our representation. Attributes are used to characterize features like
  
  - size, costs, reaction time, reliability, power dissipation for hardware units
  - storage usage for machine program and data for software units

- **aggregation** of classes or instances is provided by the has-parts- and contains-relations. However, at present our relations have to be acyclic prohibiting a recursive usage of aggregations like described in [9]. This restriction will be canceled soon since the recursive specification of hardware structures is quite an accustomed technique (cf. [6, p. 150]) for modeling generic components.

- **inheritance** is modeled by the is-a-relation. Inherited properties of a class are
  
  - behavior: If X is-a Y, the visible behavior of X at all ports present in Y is the same as the behavior of Y at these ports.
- attributes: All attributes of the "super-class" are inherited by the subclass.
- has-parts: In general it is suitable to be able to define common parts of different specializations in the base class already. The overloading mechanism allows to remove an inherited part by simply overloading it with a corresponding part of arity 0.

- encapsulation: A class in the conceptual hierarchy is defined by its attributes, the classes it is connected to, and the behavior it represents. This means that the implementation of each class in VHDL can be changed at any point without any consequences to the conceptual hierarchy.

- overloading: Attributes and has-parts-connections can be overloaded, i.e. the (default) values of each attribute can be changed in a derived class, the arity of the has-parts relation, too. With this concept, software units can be prevented from inheriting the has-parts relations of their ancestors by simply overloading them with a has-parts of arity zero.

- abstract classes: A class can be used as an abstract class. In this case, it will just define a set of attributes and a certain behavior without implementing anything and can be used as a base for deriving several different specialized classes.

4 A textual representation

In order to combine the conceptual hierarchy and the VHDL descriptions of the behavior of each class into a single framework we introduce a notation for classes and instances called CLINT.

A conceptual hierarchy specified in CLINT is defined by a set of class declarations. For each class, the declaration referring to it encapsulates the externally visible behavior while all details on how this behavior is realized are hidden in the views associated to the class.

The two central concepts in CLINT are the class definition and the instance definition. A class definition can include declarations of a set of attributes, ports, and parts the class contains as well as information about its super-class and the way it is derived from this super-class. Any given class hierarchy can be represented by a set of class declarations.

An instance is declared by an instance definition containing similar constructs: Attributes present in the class can be overloaded by attribute definitions, the parts this instance contains can be provided. A set of instance definitions describes an instance tree like a specification or a construction.

5 From CLINT to VHDL

A set of class declarations given in CLINT can easily be transformed into a set of VHDL files. For each class C one file is generated containing

- an entity declaration derived from the class definition.

algorithm compile
input: a function mapping classes to their definitions,
  \( f : \text{class} \rightarrow \text{class}_{\text{def}} \)
a class C
output: a VHDL-file \( f \).

(* compute all attributes and parts C defines or inherits *)
\[
\text{let } \text{attr} = \emptyset \text{ be a set of attributes,} \\
\text{port} = \emptyset \text{ a set of ports.} \\
\text{for each class } C' \text{ with } \forall n > 0 : \text{C is a } C' \text{ do} \\
\text{attr} := \text{attr} \cup \text{attributes}(f(C')); \\
\text{port} := \text{port} \cup \text{ports}(f(C')); \\
\text{od;} \\
\text{let gen be the set of generic attributes in attr;} \\
\text{let } \text{arch be the architecture body for C;} \\
\text{f = f } \cup \text{ arch;} \\
\text{return } f ; \\
\]

Figure 3: The algorithm to compile a CLINT class definition into VHDL code

- a "special"-architecture body for each subclass of \( C \) mapping a component of class \( C \) onto a component of the subclass by a component instantiation statement
- a "behavior"-architecture body if defined by the view declaration.

All information needed to generate a VHDL file containing these entity declarations and architecture bodies for a given class \( C \) is available in \( C \)'s definition and the definitions of \( C \)'s super-classes. Thus, an algorithm to transform a class declaration into VHDL can be implemented in a quite straightforward manner. Fig.5 shows an informal algorithm for this transformation. In addition to generating VHDL files the algorithm has to perform several checks in order to be able to reject incorrect class descriptions.

6 The model generator

In the previous section an algorithm was presented generating a set of VHDL files for a class hierarchy, one file for each class in the conceptual hierarchy. Now we show how a model generator transforms these files into VHDL code reflecting the behavior of one specific construction based on these files. The objective of this model generator is to produce a prototype of the
The target system that can be simulated and synthesized with tools like Synopsys.

The model generator expects the behavior-architectures to implement the has-parts relation in a certain way:

- Instances that are the sole part of a given role are accessed with a component instantiation statement,
- if \( n \) instances are sharing a role their component instantiation statements are contained in a for .. generate statement

Basically, the model generator works using the following scheme: After having read the instance definitions and transformed them into an instance tree it starts with the topmost component in this tree, \( c \), of class \( C \). It then generates a VHDL model for \( c \) by

- recursively generating models for the subtrees rooted by the parts of \( c \)
- changing the VHDL file for \( C \) according to the generic parameters and the name of \( c \).

The VHDL file for class \( A \) uses a component instantiation statement to insert an instance of class \( X \). However, the instance tree shows that \( A \) has to use the specialization \( Y \) instead of an \( X \) here. The model generator has to substitute the component instantiation statement by another one usable for components of class \( Y \). The information on how this substitution can be achieved is available in the special architecture of \( X \) that describes how a specialization from \( X \) to \( Y \) takes place. The substitutions that have to be performed are shown by dashed arrows in the figure. Sequences of specializations are handled by sequences of such substitutions.

Moreover, the model generator has to distinguish two cases: If only one instance of a role is present the corresponding component instantiation statement has to be modified; if several instances with this role are given a for <index> .. generate loop exists and all occurrences of the index variable in the component instantiation statement contained in this loop have to be substituted by \( i \) when generating the \( i \)-th part, the loop itself has to be eliminated.

7 Example

A concluding example will demonstrate how system design with our approach works. The basis of this example is another extract of our function hierarchy for RISC processors representing flat register files with two read ports and one write port. Fig. 5 shows this class hierarchy. The behavior of every class has been implemented by the designer as a VHDL architecture body.

Suppose the initial construction derived from the specification would consist of a single REGFD_R0 and set SIZE to 4, i.e. the register file should contain 4 registers. The configuration system would generate the construction depicted in Fig. 6. Note that a register file of type REGFD_R0 contains only SIZE−1 parts.

![Figure 4: Task of the model generator](image)

![Figure 5: An example: Extract from our function class hierarchy](image)
function class REGFD isa REGD is
  part (REGFILESIZE REGGATED);
end REGFD;
...
function class REGBASIC is
  attribute (WIDTH: generic integer := 8);
  port (  
    INPUT: in std_logic_vector(WIDTH...  
    OUTPUT: out std_logic_vector(WIDTH...  
    RESET: in std_logic;  
    CLOCK: in std_logic);
end REGBASIC;

And the instance tree is be transformed into the CLINT instance definitions

instance REGFD_0_0: REGFD_0
  part (REGGATED_0; REGGATED_1; REGGATED_2);
end REGFD_0_0;
...

The class definitions are transformed into

- a set of VHDL entity declarations — one for each class definition
- a set of VHDL architecture bodies — one for each is-a arc.

For a REGFD, a file containing an entity definition with generic parameters corresponding to the generic attributes and an architecture body containing a component instantiation statement from REGFD to its only specialization, REGFD_0 is produced.

Now, the model generator has to perform the following tasks for REGFD_0_0:

- Generate the models for all parts of REGFD_0_0.
- For every instance, produce a copy of all VHDL definitions concerning its class and substitute names appropriately.
- The architecture body of REGFD_0 contains a for ... generate statement for SIZE components of type REGGATED. This statement has to be unfolded into three component instantiation statements for the three instances. In the component instantiation statements in the VHDL files for the instances of REGGATED the class names have to be substituted by instance names.

- Wherever the REGFD_0 is used as specialization of a REGFD, the corresponding component instantiation statement for a REGFD has to be substituted by one for a REGFD_0. In this case this is trivial since both classes use the same interface.

The prototype produced by the model generator consists of one VHDL file for each instance found in the construction. After feeding these VHDL files bottom-up into tools like Synopsys the prototype can be simulated or synthesized (if the designer used the synthesizable subset of VHDL when creating his architecture bodies).

8 Conclusion and future research

We have presented a new method to design digital systems of a given domain in VHDL based on high level specifications. Features of our method are an informal specification that makes it very easy to specify the system to build, a rapid design cycle making changes to the system being designed very easy, and automatically generated prototype descriptions that can be simulated and synthesized.

Future work will mainly concern a knowledge based configuration system assisting the user in finding specifications satisfying given constraints. An outline of this system has been presented in [10]. Another major part of our research will be directed towards making other views such as Petri-nets available. This would enable us to analyze different aspects of the system's behavior. Completing our class hierarchy for RISC processors will be another central goal. Furthermore we are working on a formal description suited for model checking to automatically prove certain properties of a system.

Our work is supported by the “Deutsche Forschungsgemeinschaft” under project VF 1298.

References


