An Object–Oriented Model for Specification, Prototyping, Implementation and Reuse

Jörg Böttger, Karlheinz Agsteiner, Dieter Monjau, Sören Schulze

Department of Computer Science
Chemnitz University of Technology
09107 Chemnitz, Germany
Email: Joerg.Boettger@informatik.tu-chemnitz.de

Abstract

This paper presents a hierarchical, object–oriented model as a basis for reuse of components in the design process of digital systems.

The model forms a uniform knowledge base which consists of formal descriptions about functional, qualitative, and quantitative properties of systems and components. It supports the synthesis of systems from the described components. Starting at a system specification different models and descriptions are generated for simulation, prototyping, analysis and high level synthesis.

1 Introduction

The reuse of components at different design levels is an important basis for a rapid, inexpensive, and correct design of complex systems.

The works [3][4][5] deal with methods to reuse VHDL component descriptions for simulation and high level synthesis. They use the concept of generics to parameterize VHDL descriptions. The actual parameters are set by type declarations and conditional, loop, and generate statements. Via these parameters data width, functionality or component structure are controlled. But the exclusive use of the parameter concepts causes the number of parameters for larger systems to increase. The results are very complex and unreadable VHDL descriptions.

[1] presents a Module Manager as an approach to assist the designer in specification of hardware systems. This flexible expert system proposes behavioral solutions at a high level of abstraction. The Module Manager allows reusing them. It aims at assisting the designer with a knowledge base to generate a set of behavioral models (e. g. speedCHARTS, VHDL) corresponding to the requirements definition. For representing knowledge a semantic net consisting of patterns of interconnected nodes (classes) and arcs (relationships) is used. There are four types of association to represent all possible configurations for modeling designs: generalization (is-a), aggregation (has-a), restriction (can-be) and possibility (may-have).

In a case study [7] a simplified model of a MC68000 microprocessor has been developed by applying the Object Modeling Technique OMT by Rumbaugh ([6]) in combination with an object–oriented language extension of VHDL. The system is modeled by three different views: object model, dynamic model and functional model. The object model shows the static structure of the system. It consists of entities (e. g. execution unit, internal memory, interface) and their relations to each other. These entities are the classes of the system and its objects, respectively which are described by their attributes and operations. Relations can be aggregation, association, or generalization. The dynamic model describes changes of objects as a result of responses to interactions between objects. Finally, operations are described by the functional model (data flow diagrams, pseudo–code or code in the implementation language, an object–oriented extension of VHDL).

In [2] a generic processor model including a generic load description for performance analysis purposes is described. A processor model is specified by an instruction set and address modes. Its structure is described by a generic, hierarchical CPU tree that bases on hardware modules connected by “specialization/implementation” relations (inheritance hierarchy) and “consists of” relations (collection of models from function blocks of the layer beneath).

In [8] a methodology based on a hierarchical model of interpreters is presented for formalizing RISCs in general. The purpose of this work is the development of a generic methodology for the hierarchical specification of a large number of cores of realistic RISC processors. The abstraction levels used by a designer when implementing RISCs are mirrored by this hierarchical model. The informal specifications given by the user at each level of abstraction can

easily be converted into a formal specification in higher order logic. The model is of great use in formal verification.

Some of the literature [3][4][5] deals with generic, parameterizable component descriptions. This concept is suitable for systems with a complexity of e. g. multiplier, register files, or simple microcontrollers. The exclusive use of the parameter concept for designing complex systems implies large and hardly manageable parameter lists and unreadable component descriptions. Other works [1][2][8] are limited to a specific processor or specific views of processor descriptions. Object–oriented and knowledge–based techniques are applied in these cases.

Our model is suitable for designing complex components and subsystems at system level (e. g. for the domains of RISC processors or robotics control). All possible components and system architectures of a given domain are formally described in an object–oriented class hierarchy (knowledge base). The classes represent comprehensive information and views of components, i. e. interfaces, behavior descriptions for different purposes (simulation, high–level synthesis, verification) in different languages (VHDL, C++, temporal logic), attributes for parameters (data width, memory size), qualitative and quantitative properties (area, power consumption) and additional design constraints. The selection of components and the integration into subsystems is performed by methods of knowledge–based configuring. Model descriptions of the designed system in a specific language, e. g. VHDL, are assembled by model generators from the behavioral descriptions of the classes participating in the model. The configured systems are estimated by analysis methods and compared to the given specification requirements. An incremental extension of the knowledge base can be realized very easily by the object–oriented nature (encapsulation, inheritance) of our class hierarchy.

In section 2 our object–oriented model is described more in detail. Section 3 introduces the design methodology and the developed tool WISYRA, an example follows in section 4.

2 Model

2.1 General model structure

The model consists of an object–oriented conceptual class hierarchy and conceptual constraints. The class concept is similar to the classes known from object–oriented analysis and design but include special features for hardware design. Each class represents a pattern for constructing new objects (instances) of this class. A class defines at least a set of attributes that characterize quantitative or qualitative properties of the class as well as optional descriptions of interface, behavior, architecture or implementation of the class (figure 1). The different descriptions of a class represent different views upon this class. Accordingly different kinds of descriptions (e. g. for simulation, synthesis, verification) are used.

```
class <identifier>
   isa <identifier>         -- link to superclass
   attributes ...           -- set of attributes
   constraints ...          -- set of constraints
   parts ...                -- link to subclasses
   view                     -- one view, e.g. VHDL
      interface
         method in ...
         method out ...
         port in ...
         port out ...
      end interface ...
      source ...            -- link to source code
   end view
   view ...                 -- another view, e.g. C++
end class
```

Figure 1: Structure of a class

The conceptual hierarchy is partitioned into three layers: a specification layer, a function layer and an architectural layer.

- The *specification layer* captures knowledge about specifying a target system to be constructed in terms of functional, quantitative, qualitative and structural (architectural) requirements.
- The *function layer* contains knowledge about the functional properties the domain can consist of. Essential parts of classes in this layer are executable behavior descriptions including their interfaces.
- The *architectural layer* represents knowledge about possible implementations of system functions by architectures (consisting of software and hardware components and connections between them). These implementations can be regarded as different realizations of subsystems or whole systems at different design levels (e. g. synthesizeable VHDL code, compilable C++code, test benches, input descriptions of layout generators).

Prototypes are represented by class instances connected by an aggregation or implementation relation. They are constructed by applying the knowledge about dependencies between the specification and function layer to a given specification. Correspondingly system architectures can be constructed by exploiting in– or between–layer relations of the function and architectural layers.

2.2 Classes, objects and relations

Each concept representing knowledge about the domain is modeled as a class. Classes are distinguished by unique identifiers. A class represents a pattern for constructing concrete objects (instances). Every object belongs to exactly one class.

A class defines a set of attributes that characterize its quantitative or qualitative properties. Instances inherit the types as well as the default values of these attributes but may overload the default values. Typical properties represented by attributes are generic parameters (e. g. the width of a register or the number of registers in a register file) and other quantitative (e. g. maximal area or costs, reaction time) as well as qualitative data (e. g. fairness requirements in dealing with non determinism).

In addition to attributes, a class may define generic descriptions of interface and behavior in algorithmic or descriptive form by different views, e. g. by VHDL, C++ or temporal logic (hybrid modeling depending on the intended usage). When a certain type of descriptions is needed by the designer or the configuration system the respective view is requested from the knowledge base. A typical example: the configuration system has to evaluate the utilization of a certain subsystem. The respective analysis method will initiate a simulation of the corresponding object substructure. After reading the VHDL views of all instances participating in the substructure a model is built and simulate.

```
function class SYSTOLIC_ARRAY isa MAT_MULT_UNIT is
  comment ("Systolic Array for Matrix Mult.");
  attribute (
    DIM : generic integer := 4;
    OVL : generic integer := 2 * IVL + log(DIM,2) );
  view vhdl
    source ("SYSTOLIC_ARRAY.vhd" simulation);
    interface (
      LEFT : in std_log_vec(DIM*IVL-1 downto 0);
      RIGHT : in std_log_vec((2*DIM-1)*IVL-1 downto 0);
      OUTPUT : out std_log_vec(DIM*OVL-1 downto 0) );
  parts (
    MULT_ARRAY : DIM * (2 * DIM - 1) SYSTOLIC_CELL );
end SYSTOLIC_ARRAY;
```

Figure 2: Functional class

Figure 2 shows a part of the domain of matrix multiplier at the function layer in our class and instance notation (CLINT). CLINT was developed for an easy and comprehensive description of the object–oriented database and the generated instance nets. The important parts of a class: *attributes*, *views*, *interfaces*, specialization (*isa*), and decomposition (*parts*) are highlighted.

Classes are patterns for constructing objects, which are used to model concrete components in the domain. A target system is modeled by a hierarchy of objects which are connected by relations. This object model has to satisfy all given design requirements. A class hierarchy emerges from connecting classes by relations. Three types of relations are available to connect classes and build hierarchies:

- an *is-a* relation to describe specializations within a layer (taxonomical hierarchy),
- *has-parts* relation to represent aggregation within a layer (decompositional hierarchy),
- an *implementation* relation to model the mapping of classes of the specification layer into classes of the function layer or classes of the function layer into classes of the architectural layer, respectively.

is-a is an $1{:}n$ relation. Every superclass may have different specializations. Specialization means particularization, i. e. refinement and/or extension of attributes, interface or behavior of a class.

has-parts is a $m : n$ relation. Several subclasses may be related to one superclass, correspondingly several superclasses may be related to one subclass.

The actual design of class hierarchies by *is-a* and *has-parts* is left to the domain expert designing the hierarchies. Usually, though, *is-a* relations are more common at the function level than at the architecture level, *has-parts* are more usual at the architecture level than they are at the function level.

The implementation relation is quite different to the other two relations. While these both are strictly *intra-layer* relations, implementation only occurs as an *inter-layer* relation. Between the specification and function layer implementation means implementing specification concepts by functions. Between function and architecture layer it means implementing functions by architectures. In the general case this relation is $m : n$ between classes and attributes of both levels, in some situations more complicated relations are required to map e. g. specification concepts to functions like the instruction decoder of a RISC processor ([11]).

2.3 Constraints and constraint net

Conceptual constraints represent non–hierarchical dependencies between attributes of classes or their instances, respectively as well as between the existence of instances of certain classes. For example, constraints between a class and its successors in the *has-parts* relation are used to fix dependencies between attributes of the respective instance and its parts.

A constraint is defined by a pattern consisting of a substructure of a class (pattern), an expression that has to be

true whenever the pattern matches an object substructure, and finally a relaxation factor that specifies how “hard” the respective constraint is. When generating an object structure a constraint net is automatically constructed from those conceptual constraints if the pattern match the object structure. By propagating the constraint net of an object structure inconsistencies in the structure are revealed if the net is not free of conflicts. The conflict is the more serious the higher the relaxation factor of the constraints involved in the conflict is.

```
specification constraint IVL_DIM_OVL is
  pattern P0 (
    N0 :  any ARCHITECTURE;
    N1 :  any INPUT_VECTOR_LENGTH -> N0;
    N2 :  any OUTPUT_VECTOR_LENGTH -> N0;
    N3 :  any MATRIX_DIMENSION -> N0);
  relax (0.5);
  expr (N2.OVL >= 2 * N1.IVL + log(N3.DIM, 2))
end IVL_DIM_OVL;
```

Figure 3: Conceptual constraint

The example in figure 3 shows a part of a class hierarchy describing the domain of matrix multipliers (example in section 4). The constraint expresses the relation between matrix dimension and the data width of the values of input and output matrices, if all three are parts of the same architecture (N0). The minimal output vector length (N2.OVL) is equal to twice the input vector length (N1.IVL) plus an additional logarithmic overhead caused by potential carry flags occur during (N3.DIM) additions of subresults.

2.4 Our approach and UML

During the last years several approaches towards object–oriented modeling have emerged that provide frameworks for describing object–oriented class hierarchies. The most prominent is the Unified Modeling Language (UML) [9]. UML defines “static structure diagrams” that include nearly all notions defined by our knowledge model. Figure 4 depicts the concepts required for representing knowledge about digital systems in an object–oriented hierarchy and the way they can be described by UML. While UML offers a pendant to every notion defined by our approach (and much more) it can not replace our domain model because of several essential advantages that our approach offers:

two-layer interface: UML is targeted at supporting the development of software tools. Thus its class descriptions only offer software interfaces as sets of methods. Our approach provides two layers of interfaces: a port interface, ie. a set of in/out ports, and a set of method interfaces on top of it. Hardware classes contain at least the port interface and typically additional method interfaces as protocols on these ports, software classes typically provide only method interfaces.

integrated constraint system: UML supports constraints only as an informal comment — checking them is “tool responsibility”. Our domain model explicitly includes a full–featured constraint net that can be propagated, checked for conflicts, that can be used to update attribute values of instances, and so on. Thus non–hierarchical dependencies that are outside the scope of UML are handled in a formal manner in our approach.

three distinct layers: Our domain model consists of three different layers of knowledge that are connected by powerful mapping relations. In this way the whole design flow of a system beginning with specification and ending with implementation of system functions by architectures is covered by the domain model. In UML it is left to the user to create such a model.

more universal decomposition relation: UML's association relations only allow association roles with a fixed multiplicity. In our domain models the multiplicity of a decomposition relation can be formulated as an expression on constants and attribute values. This enables the designer to conveniently model structural dependencies like a register file that consists of as many registers as an attribute it defines describes. This kind of dependency is common in hardware description and an important feature for an appropriate representation of hardware domains.

views and model generators: Our view concept allows to independently describe the behavior of classes by e. g. VHDL, C++, and any other language. Model generators take care of mapping an object–oriented model of a system to not necessarily object–oriented target languages like VHDL. UML, on the other side, is explicitly focused on object–oriented target languages and unable to handle eg. VHDL code generation.

3 Design methodology and tool WISYRA

The goal of our approach is a comprehensive design support of systems in a specific domain by using (reusing) existing component descriptions and the knowledge about the domain.

Foundation is the analysis of the target domain and construction of a knowledge base containing information about concepts and relations of the designable systems. Currently we have implemented a relatively simple domain, matrix multiplier, and a complex domain, RISC processors.

Starting point of a new design is a set of functional, qualitative, and quantitative requirements of the system to be

notion	expressed by UML	expressed by our approach
class	class	class
parameterized class	template	generic attributes
attribute	attribute	attribute
inheritance	generalization	specialization
aggregation	association/composition	decomposition
constraints	constraints (informal)	constraints (formal)
interface	operation	port, method
class behavior	interact./state trans. diagram	view (e. g. VHDL, C++)

Figure 4: Object–oriented concepts expressed by UML and our approach

designed. These requirements are interactively mapped by our specification editor onto a set of objects, relations, and constraints of our class hierarchy at the specification layer using the knowledge base. The model is completed by the configuration system and checked for inconsistencies. This model at specification layer is transformed into different models at function and architecture layer. Depending on the information in the knowledge base the transformation is done automatically or can be controlled by the designer. The models of the target design are estimated using the analysis information about components and subsystems in the knowledge base. Also the models can be written out in different descriptions and simulated, tested, and analyzed using other commercial tools. Model generators for different languages (e. g. VHDL, C++) automatically create a complete description of the modeled system from the *view*s of the object descriptions and the knowledge about connection between the included components.

At least a synthesizeable VHDL description of a model at architectural layer can be generated.

This methodology and a corresponding tool set is being developed in the project WISYRA. Essential parts are:

- knowledge based configuration system
- constraint system
- specification editor SpecEd
- model generator for VHDL, C++
- graphical input and output of class hierarchy and designed models

Interfaces are implemented connecting external commercial tools (e. g. tools by Synopsys) for simulation, synthesis, or analysis.

4 Example

By the example of matrix multipliers we show the strength of our approach and the developed tool WISYRA. Basis of the design is the description of the domain using object–oriented class hierarchies. These conceptual hierarchy at specification, function, and architecture layer are the knowledge base for the design by WISYRA.

The example is a matrix multiplier for $n \times n$ matrices. A matrix multiplier can be specified by matrix dimension, word width of matrix values, maximum chip area, or the principle kind of computation (each matrix value sequential, a row/column parallel, or a special highly parallel way using a systolic array).

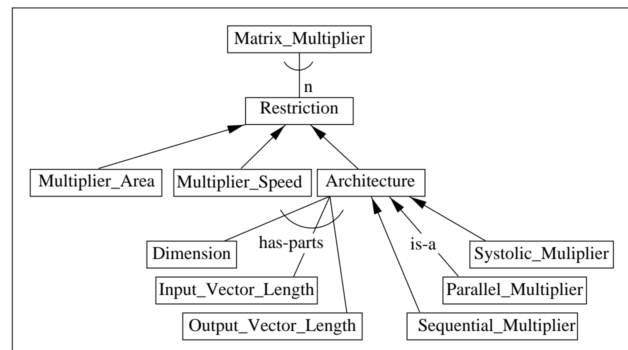

Figure 5: Class hierarchy at specification layer

This paragraph presents the design flow from a set of requirements to a synthesizeable VHDL description: The designer chooses the class Matrix_Multiplier of the specification hierarchy (figure 5) as starting a point of the design and creates an instance of this class. The specification editor SpecEd (figure 6 shows one of the interaction windows) asks the designer to fill in all requirements to specify the target system.

The configuration system that runs in the background supports this process by using and evaluating the knowledge about structures and dependencies of the knowledge base.

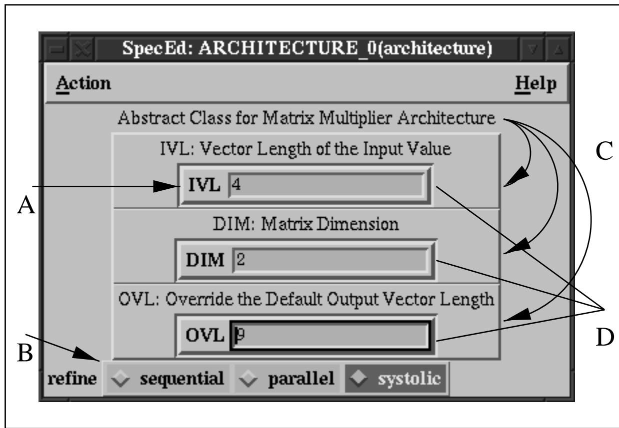

Figure 6: Specification editor specifying a matrix multiplier

Figure 6 shows different kinds of inputs and dependencies using SpecEd. Point A represents the input of the attribute value *IVL* of the object Input_Vector_Length. Furthermore the architecture of matrix multiplier can be specialized to a sequential, parallel, or systolic matrix multiplier (point B). This means that in the knowledge base a specialization relation (*is-a*) exists from class Architecture to the classes Sequential_Multiplier, Parallel_Multiplier, and Systolic_Multiplier. Point C shows the automatically partitioning of a component in subcomponents by our tool. The aggregation relation *has-parts* connects class Architecture and classes Input_Vector_Length, Dimension, and Output_Vector_Length in the knowledge base at the specification layer. Constraints of the knowledge base will be automatically created and evaluated. Point D shows the three attributes, which instantiate a constraint of type IVL_DIM_OVL (see figure 3) to describe the dependency between these attributes.

If the specification of the component matrix multiplier is not complete after the designers input the configuration system will complete the object hierarchy using the knowledge of the knowledge base. The result is a complete object hierarchy at the specification layer.

The next step is to transfer a freshly created model of a matrix multiplier from specification layer to function layer. The knowledge about which object and attribute at the specification layer entail objects, relations, or attribute values at the function layer is part of the knowledge base. Using this knowledge some objects and relations on function layer are created as skeleton of our functional model. These objects are the instance Mat_Mult_0 of the functional class Mat_Mult including attribute values IVL=4, DIM=2, and OVL=9 and the instance Ctrl_Systolic_1 representing the control unit of a systolic matrix multiplier.

The configuration system constructs an object model at the function layer from this initial objects. This is described in section 3 and more in detail in [10]. A well defined knowledge base causes a nearly automatic configuration of the described target system. But the designer can control the creation of the model and can interact with the system.

As result an object model is built which only consist of a set of instances at function layer and aggregation relations *has-parts* between them. Then all constraints are satisfied and general objects are replaced by specializations as far as necessary, e. g. the root node Mat_Mult_0 is replaced by a specialization Systolic_Mult_0 (figure 7).

This object model can be used in two ways: It can be transformed into a model at the architecture layer using concrete implementation for subcomponents on function layer or a description of these functional model is generated from the descriptions in the views.

In the first case all leaf nodes of the functional object hierarchy are transferred into corresponding objects at architectural layer. At the lower left corner of figure 7 a concrete adder realization (carry look ahead adder : ADD_CLA_0) at architecture layer implements a functional class (addition operation : ADD_OP_6). The objects at architecture layer contain synthesizeable VHDL descriptions. A model generator builds a complete VHDL model from these descriptions. The task of the model generator is to substitute generic parameters by their values or to connect entities with their components which result from specialization. The result is a simulateable or synthesizeable description of the matrix multiplier.

In the second case different model descriptions can be created from the object model at the function layer. As described in section 2.2 every class can contain one or many different views of the class. Each view represents a description of the class in a specific language for a specific purpose. A model generator of this language creates a model of the component in this language. The model generator of VHDL and C++ can generate a model description of the matrix multiplier.

5 Conclusion

A hierarchical, object–oriented model was presented as a basis for designing digital systems. It allows the construction of knowledge bases for various domains of application. A knowledge base represents knowledge about specifying systems by functional, quantitative and qualitative requirements, about system functions and subfunctions and about feasible system architectures. Our approach includes the option to generate functional prototypes, means of ver-

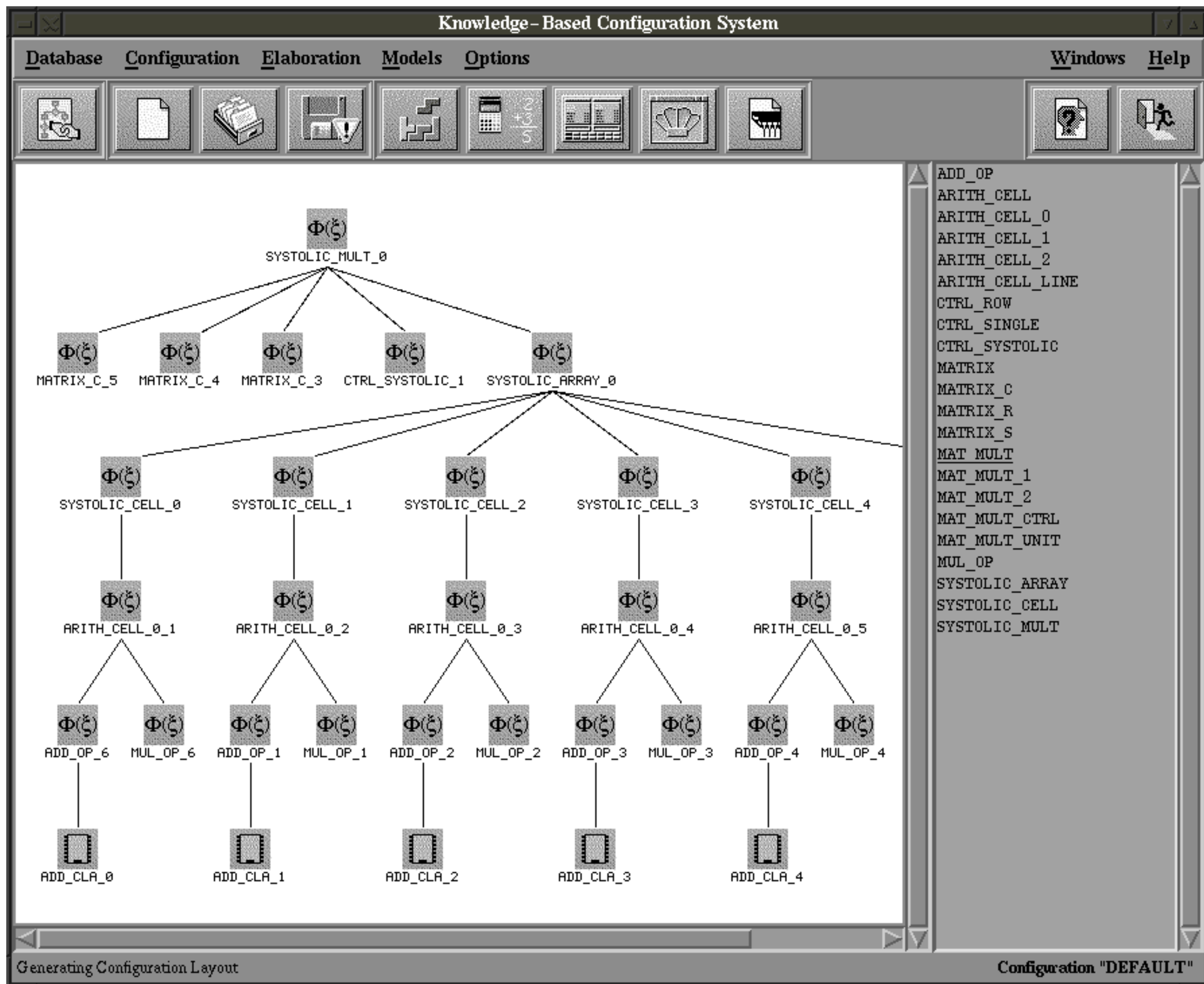

Figure 7: Object hierarchy at function layer

ification, synthesis of architectures, testing and reuse of characteristic elements in the various design layers.

Advantages of our approach are:

- after developing the knowledge base (only once), formal descriptions can be built of many different target systems of this domain,
- the simple, informal specification of the target system by requirements,
- the reuse of specification, function and architecture knowledge including behavior described in various languages as well as of systems or subsystems designed previously,
- a quick and easy generation of prototypes for simulation, synthesis or mapping to FPGAs, and
- the systematization of knowledge and documentation of systems of a given domain.

Limitations are set by the cost of developing the knowledge base. As in all other reuse scenarios there is an overhead in modeling effort. The effort building our object–oriented class hierarchy is high but the breakthrough can be reached to amortize the modeling effort. If many different designs of one domain are made using the same knowledge base the tool WISYRA will work profitable.

Actual work is done at the area of analysis methods and for reuse of components in heterogeneous systems (hardware/software, analog/digital) using WISYRA. Furthermore additional domains will be modeled to get further

results about the application of our model and our tool.

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

References

[1] L. Chaouat, C. Munk, A. Vachoux, and D. Mlynek. An expert assistant for hardware systems specification. In *Proc. workshop on libraries, component modeling, and quality assurance*, Nantes, France, April 1995.

[2] U. Langer. *Ein Konzept zur entwurfsbegleitenden Leistungsanalyse von Rechensystemen. Dissertation*. Kovac, Hamburg, 1995.

[3] V. Preis, R. Henftling, M. Schütz, and S. März-Rössel. A reuse scenario for the vhdl-based hardware design flow. In *Proc. Europ. Design Automation Conf. EURO-DAC'95*, Brighton, Great Britain, September 1995.

[4] V. Preis and S. März-Rössel. Aspects of Modeling a Library of Complex and Highly Flexible Components in VHDL. In *Proc. Workshop on Libraries, Component Modeling, and Quality Assurance*, Nantes, France, April 1995.

[5] A. Reutter, B. Mößner, I. Kreuzer, and W. Rosenstiel. Wiederverwendung komplexer Komponenten für Synthese und Simulation unter Verwendug von VHDL. In *Proc. 8. E.I.S.-Workshop*, Hamburg, Germany, April 1997.

[6] J. Rumbaugh and other. *Object-oriented modeling and design*. Prentice Hall, Englewood Cliffs, USA, 1991.

[7] G. Schumacher, W. Nebel, W. Putzke, and M. Wilmes. Applying object–oriented techniques to hardware modelling – a case study. In *Proc. VHDL User Forum europe, SIG-VHDL'96*, Dresden, Germany, May 1996.

[8] S. Takar and R. Kumar. A formalization of a hierarchical model for risc processors. In *Proc. EURO-ARCH'93*, Reihe Informatik, München, Germany, October 1993.

[9] UML Unified Modeling Language. In *http://www.rational.com/uml/references/* 1997.

[10] S. Kahlert, J.-U. Knäbchen, D. Monjau, S. Schulze. System Level Synthesis Using Knowledge-Based Configuration Tools. In *Proc. of Workshop on Computer Aided Systems Technology*, Ottawa, Canada, May 1994.

[11] K. Agsteiner, D. Monjau, and S. Schulze. Automating system-level design: From specification to architecture. In *Proc. EUROMICRO '96*, Prague, Czech Republic, September 1996.