Generating ECAD Framework Code from Abstract Models

Joachim Altmeyer Bernd Schürmann Martin Schütze
University of Kaiserslautern
D-67653 Kaiserslautern, Germany

Abstract - This paper describes a novel approach for generating the code of ECAD frameworks automatically from abstract models. Instead of providing the framework components of a design system in the form of a code library, we favor to use such a 'library' at modeling level. The framework components, common to all design tools, are described by easily understandable base models. For each design tool, the base models will be customized individually to meet the tool-specific requirements. Then, the customized models are input to a code generator that automatically generates tool specific, optimized source code. A generic modeling system and several code generators are implemented in a software generation environment called MOOSE.

I. INTRODUCTION

VLSI design systems perform many tasks in various domains and hierarchy levels. These tasks may be synthesis, extraction, simulation, and verification steps. All actions will be performed by different tools. A complete ECAD system is a large collection of design tools together with a central data management system (e.g. a design database) and a design management system.

After more than two decades of implementing standalone tools for designing ICs, most design systems became more and more integrated. They follow the ECAD framework approach to reduce the rapidly increasing software costs. Overcoming the 'software crisis' is such important that ECAD frameworks became a notable research discipline. A good survey can be found in [1]. The goal of the framework approach is to improve both, the design of integrated circuits by providing an always consistent view of the design data as well as the development of the design tools themselves. Since data management, user interface, graphics, and control services are implemented inside the framework, the code of the tools can be reduced to the application specific algorithms. All algorithms use the same framework infrastructure. Framework components are comparable with software libraries which have fixed interfaces to the application (e.g. [2]).

At first sight, this approach seems to be the ideal solution. Nevertheless, it has some drawbacks which are caused by the inflexibility of the library concept. This is mainly true for the data management component. We therefore concentrate on the data handling during the rest of this paper.

For efficiency reasons, design tools usually cannot be implemented directly on top of a multi-user database. Their runtime data have to be stored in main memory. Input data are checked-out from the database and output data are checked-in back into the database in large units within a single transaction. To some extent, this caching technique is already provided by modern client-server databases [3], [4]. The most efficient data access for design tools, however, is still provided by local data structures which are optimized for the actual design algorithms.

The tools all regard only sections of a comprehensive data model and need similar but not necessarily identical main memory data structures which are derived from the comprehensive data model. Tool specific, local data structures should be added transparently to the structures corresponding to the common data model. The same is true for the other framework components like graphics, user interface, and control flow. All these components have some similar and some distinct parts for the different tools.

The main memory data structures are implemented as abstract data types (ADTs). In the past, they were generally hand-coded which is very time consuming and error prone. ECAD tool developers had to spend much time in providing the tool infrastructure instead of writing design algorithms.

We therefore propose a software engineering approach to get more efficient, individual but correct code. We try to generate the framework part of the tools automatically and individually from the comprehensive models. Although the final code is customized for a specific tool, the application programmer has to write the code of the application specific algorithms only. The framework components will

32nd ACM/IEEE Design Automation Conference ®

be specified at model level, i.e. the comprehensive model itself will be customized. The source code will then be generated automatically from the customized model. This process results in efficient and correct code.

In this paper, we describe a software engineering environment called MOOSE (model-based object-oriented software generation environment) which has successfully been used for generating the framework components of the PLAYOUT VLSI design system [5]. Using MOOSE we are able to generate efficient application-specific code automatically from abstract models.

The rest of the paper is structured as follows. Section 2 considers the modeling level, i.e. the level of the program specification. Section 3 then describes the generators and the structure of the code automatically generated from the models. The generated framework code of several VLSI design tools is analyzed in section 4, and we conclude the paper with some final remarks in section 5.

II. The Models

Our aim is to *model* the framework code rather than to *implement* it by hand. The precondition to do this is that these framework components can be described with rather simple, but powerful models. Within our method, the model of a certain framework component is created in three steps: Step 1 is to find a *modeling notation* for the models to be developed. In step 2 the *base model* for an application domain is developed. This is done by analyzing the domain (in this case ECAD) carefully and creating models which are suitable for the whole domain. These two steps have to be performed only once per application domain. The final model of the actual application's framework component, the *framework model*, is developed in step 3 by customizing the base model.

The MOOSE software generation environment we present in this paper consists of two parts: a generic editor to perform all three steps and a set of code generators. While the generators will be subject of section 3, we focus here on the modeling level.

A. Choosing a modeling notation

As explained above, our first step is to find a modeling notation suited to match the aspects of the application domain. For the description of data management components in the ECAD domain, we have chosen an Extended Entity-Relationship (EER) approach. This means, we enhanced the conventional ER model [6] by object-oriented concepts like inheritance and additional relation semantics as aggregation. From our experience, this notation is very well suited for data modeling in the ECAD domain and, therefore, a good starting point for the development of a powerful base model.

B. Developing the base model

After selecting a modeling notation, a highly domain specific, comprehensive model has to be developed. Its existence is a precondition for the integration of different applications in the corresponding domain. This is an observation many people working on ECAD tools and frameworks made. An early model for the description of ECAD meta data was proposed by Katz [7]. The work of the CFI on common models (e.g. design representation [2]) shows their importance for the integration of different tools into a heterogenous ECAD framework.

A base model is usually developed by combining such well-suited submodels if they together cover more or less all aspects of the application domain. It describes what is common to all applications of that domain. The design model we

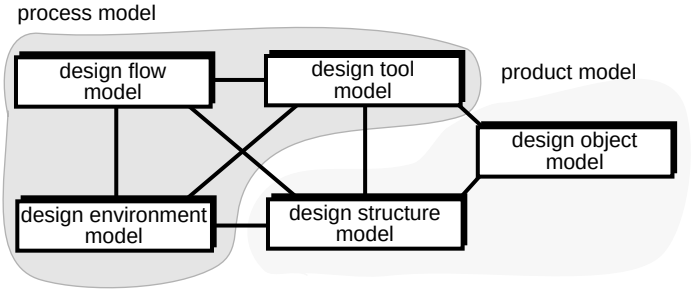

Fig. 1. PLAYOUT Design Model

use for PLAYOUT describes almost all aspects of the VLSI design process such as the design tools, the designers, and the design data [8]. It is structured into five submodels clustered in two meta models (fig. 1). The process meta model describes the involved tools, the resources (e.g. workstations), the users, and the possible design flows. The product meta model describes the design data. In this paper we will consider only the product model which serves as the base model for the generation of ADTs.

The product model is divided into the design object model describing the fine grain data for the tools (e.g. netlists, layouts, etc.), and the design structure model describing meta information such as hierarchy and versioning [9]. This approach is common to many design environments.

C. Creating the framework model

To use the base model in an application project it must be customized as it can be seen in figure 2. First of all, the base

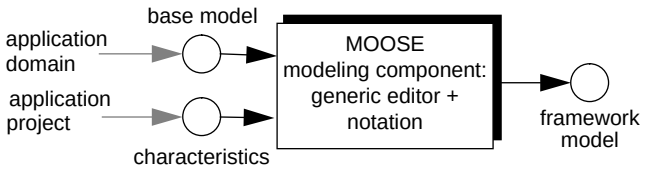

Fig. 2. Development of Models

model usually includes much more information than one application needs. It would be inefficient to include the generated code of the whole base model into the application. Furthermore, applications need local extensions to the model, for instance data structures needed for internal algorithms. They

are usually not part of the base model and have to be instantiated only for a particular application.

As explained above, the base model is a result of modeling the ECAD domain. The framework model is built from the base model by using *characteristics*. The characteristics describe the tailoring of the base model with respect to the application project. This will be explained in detail below. The MOOSE modeling component allows the user to enter the base model (once for the whole ECAD domain) as well as the characteristics (for every application project) and customizes the base model.

D. Creating models with MOOSE

Within MOOSE, base models are constructed by using the elements from the EER-notation (see above). (Sub-)models can be clustered into schemata, which in turn can aggregate other schemata, forming a schema hierarchy (see below). Different framework models can be derived from one base model by using version and configuration concepts.

The following enumeration will describe which kind of changes to the base model can be made, i.e. which types of information can be put into the characteristics (see fig. 2).

- *Schema aggregation:*
  Objects and their relations can be clustered into schemata. Schemata can aggregate other schemata. It is, for example, possible to have a schema of a tool verifying the consistency of netlists which aggregates the schema for the design structure model and the schema for netlists but not the schema for layout data. This corresponds to the view concept used in many design data models.
- *Adding/deleting object types:*
  New object types can be added to model local data structures. Object types not needed can be made invisible for the current application and will not be used for code generation.
- *Adding/deleting attributes and relations:*
  As well as object types, new relations can be added to or existing ones can be deleted from the base model. Attributes can be added to or deleted from any existing/ newly created object type.
- *Optimizing for certain purposes:*
  The data structures of the resulting framework code can be optimized. It is, for example, possible to optimize these structures for access speed or for memory consumption. In addition, object types or relations can be annotated with instructions how they can be retrieved from external data sources, resulting in an automatic data exchange with design databases or other programs (see also section 3).

The MOOSE system offers a basic technique to modify the base model with the characteristics: the configuration of different schema versions. It is possible to derive new schemata from existing ones, allowing the addition/deletion of object types, relations, and attributes as stated above. This results in new versions of these schemata. The configuration of schemata allows the tool developer to define an own application-specific framework model from the base model. The framework model is then input to the code generators.

E. Modeling experiences

MOOSE has successfully been used for the development of different tools. Table I indicates which parts of our base model can be seen by several example applications. The usage of a submodel is ranked from ‘o’ (the application does not use the submodel) to ‘+++’ (the complete submodel is used). The Chip Planner (top-down floorplanning) does not need to see the design structure model as it is not interested in versioning, the complete cell hierarchy, etc. But it needs a lot of detailed, fine-grain design data from the design object model like the shapes, sizes, and interconnections of the cells. Our design management tool needs to know the complete design structure model, including versioning, design alternatives, etc. But it is rarely interested in detailed design data (except some statistical data and information needed for graphical object representation). The Repartitioner, a tool for changing the circuit hierarchy, needs some knowledge about the design structure as well as about the design data (ports, netlists, etc.). An EDIF converter sees most of our design structure model and, more or less, the complete design object model.

TABLE I
CUSTOMIZING THE BASE MODELS FOR DIFFERENT APPLICATIONS

Part of Base Model	Application			
	Chip Planner	Design Management	Repartitioner	EDIF Converter
Structure Model	o	+++	++	++
Object Model	++	+	++	+++

These examples show that it is feasible to spent some effort in designing a good base model, because it can be reused for different applications. Customizing the base model is easy within the MOOSE system. It makes sense to customize the base model by the techniques described above as most tools don’t need the complete model but implement some additional private data structures. This approach guarantees that only highly tool-related, efficient ADTs will be built by the generator.

III. Generated Framework Components

As depicted in section 1, this paper focuses on generating data management components of ECAD frameworks. In this section, we summarize different approaches to realize the data management in existing ECAD systems and we outline the generation of these by our software generation environment MOOSE.

A. Data Management Implementation Techniques

Design systems typically have one common workspace (*archive workspace*) and several workspaces for the design tools (*private workspaces*) to manage the data of a design

transaction (see figure 3) [10]. The archive workspace stores all important design data persistently. These data may be located within a file system or within a non standard database management system. To support long design transactions, the data of design tasks are checked-out from the archive workspace (e.g. design database server) to the private workspace. During the transaction, these data are cached by the private workspace which is a client with respect to the database. Each of these clients needs its own data management to handle the cached design data. At this, because of the static property of the tool implementation, the required object types are well-known at compile-time and can be realized as an abstract data type. A query interpreter and an optimizer are not necessary for the data management at the client side. When the task is finished, the resulting data are checked-in into the database.

The exchange of data between the database server (archive workspace) and the clients (private workspaces of the design tools) can be realized by a predefined file format (e.g. [11], [12]) or by a communication transparent to the client application (e.g. via the UNIX socket mechanism) [1]. In figure 3, the different possibilities to exchange data in ECAD systems are outlined.

Only few ECAD design systems span the whole design space from behavior to masklayout. The integration of external tools must therefore be supported. Transformers to translate between the external and internal data representations must be realized.

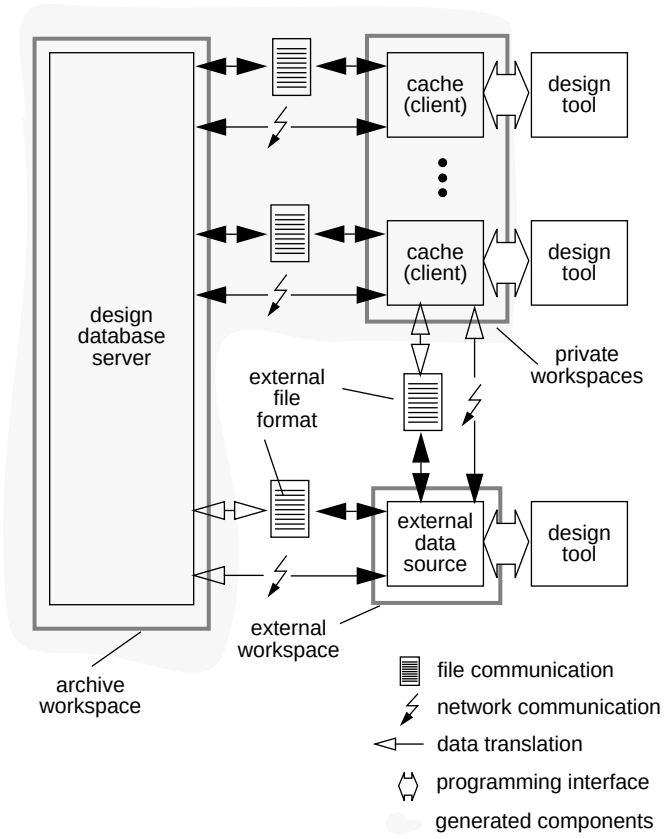

Fig. 3. Data Management of an ECAD System

With these investigations, we summarize the different requirements which should be considered by a good software generation package for data management components of ECAD systems:

- The archive workspace and the private workspaces, which cache transaction data, should be generated.
- The exchange of data between the private workspace and the archive workspace must be realized in an efficient and easy manner (e.g. by a defined file format or by network communication).
- To integrate external tools, the implementation of adequate translators should be supported.

All these requirements are solved by the MOOSE generators we describe in the following.

B. MOOSE Generator Approach

Our software engineering approach to fulfil the requirements listed above is software reuse by code generators [13]. Similar to generators like YACC and LEX the implementation of the framework code is generated from an abstract model, e.g. from an EER-model as described in section 2. Before we show the advantages of our approach, we present some MOOSE generators.

MOOSE provides different generators as shown in figure 4. The data management generators use an EER-model as input and create framework code as output. Each generator translates modeling primitives into framework code by using templates. Reuse is therefore realized in three ways: First, by using templates at each generation step. Second, abstract models are used more than once, because they are used to generate ADTs, file formats, communication, and documentation as we will see below. Third, parts of the base model are (re-)used to build the customized framework models (see section 2). The MOOSE environment currently contains the following generators for the data management.

1) C-Code Generator

Our first code generator supports the programming language C. Using a storage manager as a container library the C-code generator automatically implements functions to create, access, modify, and delete objects of each object type. It also creates functions to connect and disconnect objects, i.e. to create and destroy a relation between them. Data inheritance can be realized by copying attributes from a base object type to a derived object type. Beside these functions, routines to create savepoints and to restore saved data are available. For the application programmer the generated code serves as an abstract data type (ADT) with data management functions. To optimize the code, suitable data structures as AVL-trees or hash tables are chosen depending on the characteristic.

2) C++-Code Generator

Newer implementations of our tools use the C++-code generator which implements an object-oriented class hierarchy.

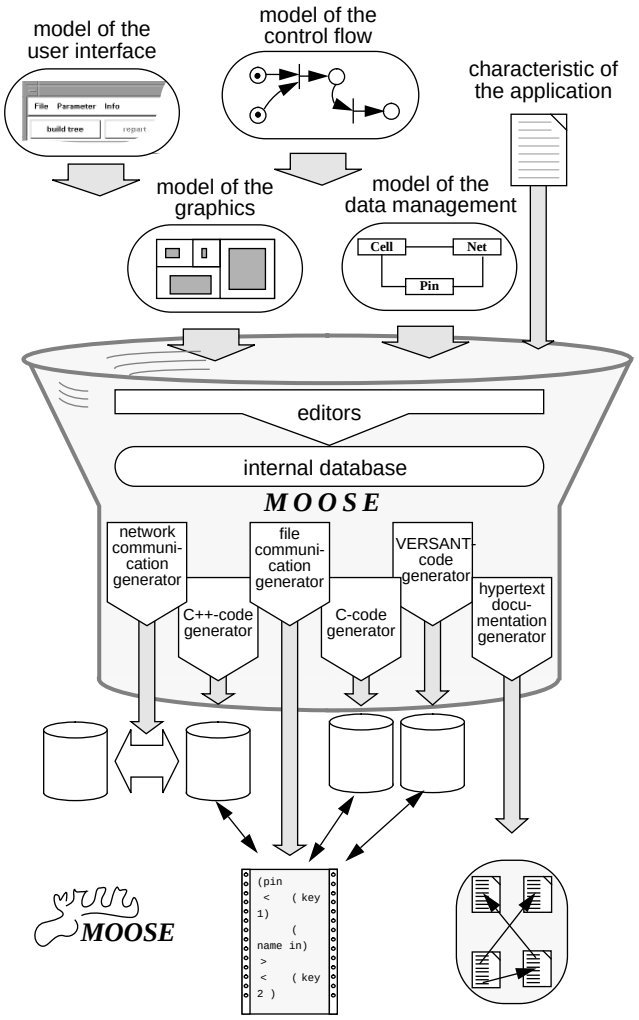

Fig. 4. MOOSE
Because this paper concentrates on the data management other generators are not shown.

For each object type it generates a class and, beside others, methods to

- construct and destroy objects,
- access and modify instance variables,
- compare objects,
- create and destroy relations, and
- save and restore the data.

The generated code uses container classes of the NIH class library [14]. The user has the possibility to extend the generated structures by adding user-defined methods to the generated classes. These user-defined methods will not be overwritten when a new class hierarchy is generated.

At this point, it is important to notice that not only the declarations but also the whole implementation code of the methods is generated. This is one of the most important differences between MOOSE which is specialized to ECAD frameworks and general purpose code generators as they are, for instance, delivered with OOA/OOD[1] systems, e.g. Rational Rose/C++[2].

3) VERSANT-Code Generator

To support persistence we developed a code generator for the data definition language of the object-oriented database management system VERSANT[3]. This code generator creates methods with the same interface as the C++-generator. But in this case, the underlying library is part of a database management system instead of main memory data structures. Because of the same interface a migration of an application from main memory management to database management and vice versa is a simple task.

So far, the code generators for the workspaces are introduced. We saw that the generated application interface is largely independent from the underlying data management system. In the following, we further address the code generators for data exchange between two generated workspaces or between a generated workspace and an external data source.

4) File Communication Generator

An ASCII file format to exchange design data between the generated workspaces is defined by generating a parser and a printer routine. Just as the ADTs, these I/O-routines and, therefore, the data exchange formats are generated from the EER model. This file format provides a communication between different workspaces within an heterogeneous network (see fig. 3).

5) Network Communication Generator

Instead of the ASCII file data exchange, a communication based on the UNIX socket mechanism can also be generated [15]. Whereas the data of the ASCII files must be checked-out and checked-in by explicit function calls, this communication is transparent to the application program. The generated private workspace then distinguishes two cases: Either, the data are already available in the ADT and can immediately be returned or, otherwise, the data must be retrieved from the archive workspace by using the generated UNIX socket communication. From the view of the programming interface, there is no difference between these accesses. Data stored in the archive workspace will be retrieved transparently to the application.

In addition to this, the network communication generator supports the integration of external data representations. As described in section 2, it is possible to integrate external systems by defining a mapping table which maps the current data model to the external view. In this table, the programmer adds necessary data queries which will be interpreted by the external data source. Of course, this implies that a query interpreter is available at the server side.

6) Hypertext Documentation Generator

To support the application programmer in using the generated ADT, a comprehensive FrameMaker[4] hypertext documentation is generated.

1. OOA/OOD: object-oriented analyze / object-oriented design
2. Rational Rose/C++ is a trademark of Rational.
3. VERSANT is a trademark of Versant Object Technology.
4. FrameMaker is a trademark of Frame Technology Corporation.

IV. Results

This section describes some advantages of our approach. We present examples of using MOOSE in our VLSI design system PLAYOUT and we show which data management components were generated.

Since we developed MOOSE, all of our VLSI design tools were (and are) implemented by using generators. Table II shows information of two design tools already mentioned above. The data management of the Repartitioner, which includes a timing analyzer, is constructed by the C-code generator using an ER-model. Our design manager DESIMA is generated by the C++-code generator based on an EER-model. In addition, DESIMA uses an object-oriented graphic library which has been referenced at the model level - not at the implementation level as it is usual. A special graphics generator enriches the generated C++-classes with a set of useful graphic routines to draw, move, delete, or highlight objects [16]. Finally, we also implemented a recent version of MOOSE by using an older version.

TABLE II
APPLICATIONS IMPLEMENTED BY USING THE GENERATOR APPROACH

Applications	Repartitioner	DESIMA	MOOSE
Model	ER	EER	EER
Language	C	C++	C++
Graphics	hand-coded	generated	generated
LOC (total)	150,000	65,000	70,000
LOC (generated)	50,000	50,000	35,000
Percentage	33%	78%	50%

The comparisons of the generated lines of code (LOC) with the total LOC number shows a generation quota up to 78%. The generated part of the Repartitioner is smaller than for the other two examples because its graphical interface is hand-coded. Altogether, the experiences of our application programmers using MOOSE are completely positive. One important factor for our success is the reduction of implementation time which can only roughly be determined. Our application programmers reported savings of about 1/3 of total implementation time.

Besides the size of the generated software, let us have a look at the generated parts of the framework by considering the requirements above. In figure 3, the generated framework components are shaded: the archive workspace, the customized private workspaces, the data exchange in form of file or network communication. Besides the SE-process described in section II, the generation of this tight-coupled client-server architecture represents one of the important differences to other interesting generator approaches (e.g. [17]). To realize the external communication management half of the communication code is generated automatically. To read and write external file formats a parser and a printer based on a generated ADT must be hand-coded (e.g. using LEX and YACC). To realize the network communication with an external data source only the part at the external site must be realized manually.

V. Conclusions

We addressed a generator based approach of automatically synthesizing efficient and correct code of ECAD framework components. We showed how we set up base models which must be done once for all tools. These base models will be customized and then used as input for our code generators. The result is automatically generated, and with that correct code of the framework components.

Due to space limitations, we restricted this paper to the description of modeling and generating data management components. Our software generation environment, MOOSE, however, is not restricted to data management. We recently completed our first generators for a graphical interface based on the X library [16] and for a Motif-based control interface. We further investigate which parts of the still hand-coded components can further be replaced by our approach. Our goal is to generate as much as possible of the application.

References

[1] T.J. Barnes, D. Harrison, A.R. Newton, R.L. Spickelmier, "Electronic CAD Frameworks," Cluwer Academic Publishers, 1992

[2] "Design Representation Programming Interface: Electrical Connectivity," Version 1.4.0, CFI Document dit-92-S-1, 1994

[3] Versant Object Technology, "C++/VERSANT Reference," Version 1.7, June 1992

[4] P. Butterworth, A. Otis, J. Stein, "The GemStone Database Management System," Communications of the ACM, pp 64-77, October 1991

[5] G. Zimmermann, "PLAYOUT - A Hierarchical Design System," Information Processing 89, G.X. Ritter (ed.), Elsevier Science Publishers B.V. (North Holland), IFIP, 1989

[6] P.P. Chen, "The Entity-Relationship Model - Toward a Unified View of Data," ACM Transact. on Database Systems, Vol. 1, No. 1, 1976

[7] R. H. Katz, "Information Management for Engineering Design," Surveys in Computer Science, Springer Verlag, 1985

[8] C. Huebel, D. Ruland, E. Siepmann, "On Modeling Integrated Design Environments," Proc. Int. European Design Automation Conference, 1992

[9] B. Schürmann, J. Altmeyer, M. Schütze, "On Modeling Top-Down VLSI Design," Proc. Int. Conference of Computer Aided Design, 1994

[10] R. H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," ACM Computing Surveys, December 1990

[11] D. Perry, "VHDL," McGraw Hill, 1991

[12] "EDIF, Electronic Design Interchange Format," Version 200, EDIF Steering Committee, 1987

[13] R. Prieto-Diaz, "Status Report: Software Reusability," IEEE Software, May 1993

[14] K. E. Gorlen, S. M. Orlow, P. S. Plexico, "Data Abstraction and Object-Oriented Programming in C++," John Wiley & Sons, 1990

[15] M. Schütze, B. Schürmann, J. Altmeyer, "Generating Abstract Datatypes with Remote Access Capabilities," IFIP WG 10.5, EDAF working conference, Gramado, Brazil, 1994

[16] S. Queins, J. Altmeyer, B. Schürmann, "Model-Based Reuse of Object-Oriented Graphic Components for CAD Systems," Proc. 39th International Scientific Colloquium, Ilmenau, 1994

[17] A. Bredenfeld, "A Generator for Graph-Based Design Representations," , IFIP WG 10.5, EDAF working conference, Gramado, Brazil, 1994