Towards Support for Design Description Languages in EDA Frameworks*

Olav Schettler†, Susanne Heymann
{schettler, heymann}@GMD.de
GMD, D-53754 Sankt Augustin, Germany

Abstract

We report on a new framework service for design tool encapsulation, based on an information model for design management. The new service uses generated language processors that perform import and export of design files to and from a design management database with the support of nested syntax specifications and extension language scripts. Our prototype design environment is based on the Nelsis CAD Framework and several tools from the Synopsys high-level synthesis and simulation tool suite.

1. Introduction

Successful cooperative design of complex systems requires the maintenance of a complex web of structural information (relationships and annotations) on design objects [5]. Several researchers [1,2,3,4,10,14] have pointed to the most important relationships: *hierarchical composition*, *version derivation*, and *equivalence* between design objects. In addition, design objects may be annotated with *level of abstraction*, *view type*, *owner*, or *status* [9,15]. Electronic design automation (EDA) environments maintain this structural information in a dedicated design data management (DDM) component [6] and keep it accessible, up-to-date and consistent with regard to actual design object representations.

Competitive designs often not only require standard tools but utilize tools developed or customized for a specific design task. EDA environments must be open to incorporate such specialized design tools. Today's open EDA environments are constructed from framework components which provide design data management and other services, and design tools which perform the actual design steps. An important objective of frameworks is to provide interoperability among the tools and between the tools and the DDM component [7]. However, after five years in existence, the design representation programming interface (DRPI) defined by the CAD Framework Initiative (CFI), the dominant standardization body in the field, is still restricted to representing connectivity data. Connectivity is only one aspect of a complete electronic design so the DRPI has to be supplemented by other means of design description and access.

Already in 1987 the hardware description language VHDL became an IEEE standard [13] and has become a driving force in EDA interoperability today. Based on sequential files, however, designs described in VHDL are not accessible without syntax processing. As a prime goal of EDA frameworks is to "facilitate cost effective, efficient, seamless incorporation of tools into design systems" [7], a framework needs to provide a service to efficiently process design languages in general and VHDL in particular to bridge the gap between file-based design reality and the promises of integrated EDA environments.

In this paper, we describe such a service. Based on a design methodology, we develop a suitable information model for design data management and show how to map it to the schema of the Nelsis CAD Framework. We then show how to process design files for import and export to and from a DDM component with the support of nested syntax specifications and extension language scripts. Finally, we describe the implementation of a prototype design environment for high-level synthesis based on the Nelsis CAD Framework and Synopsys synthesis and simulation tools and present our conclusions.

2. A VHDL-based design methodology

We assume a design methodology that proceeds basically top-down. Design starts with a functional description of a design unit on a high level of abstraction. This description defines the top-level behaviour of the design in terms of programming language constructs like communicating processes, (recursive) procedures, loops, abstract data types, or variables. The design interface is defined in terms of high-level data types.

The design implementation is then partitioned into a structural hierarchy of functional blocks, each of which is again defined by its behaviour. The functional blocks do

*. This research was supported in part by the commission of EC under ESPRIT contract 7364

†. Olav Schettler is now with ASSEM AUDI+CO GmbH, Feldstraße 8, D-53340 Meckenheim, Germany

not need to have distinct interfaces when reuse is not an issue for them. On each level of abstraction, first a partitioning into blocks with well-defined interfaces is defined and only then are the individual block implementations specified. A block implementation is transformed from the functional domain to the structural and physical domains (or "view types"), either manually or by design tools. This strictly top-down approach is particularily useful when designing in a team.

An electronic module is a design unit that realizes a specific electrical behaviour. This behaviour can only be controlled and observed through its interface. An interface has a number of interface elements which are used to interface the module with the outside word. The behaviour on a very high level of abstraction (i.e. on system level) is reflected in a core interface. The more refined the specification becomes on lower levels of abstraction, the more the data types for interface elements are refined [14]. In addition to data type refinement, there may be completely new interface elements on lower levels of abstraction, for example:

- alternative clocking schemes like single as opposed to 2-phase overlapping clocks
- different power line requirements due to different technologies (e.g. +12V, +5V, -5V as opposed to single +5V)

Moving to a different domain, however, will most certainly completely change the interface, simple because the interface primitives are different. Related to the greater difference in the kind of information represented, especially modules in the physical domain may have interfaces non-isomorphic with those in the functional and structural domains. Following this observation, we have to refine our notion of modules and state that a module may have more than one interface associated with it. These interfaces of one module are not identical but rather related to each other in an inheritance tree, where specializations have refined interfaces (cf. Figure 1).

While exploring design alternatives for a complex module, the selection of a specific implementation of a submodule needs to be flexible. During early design stages, it is appropriate to always select a working implementation of a component by some default selection rule. Depending on the design task, however, the explicit selection of implementations from different levels of abstraction and from different domains may be more appropriate. Alternatively, the designer of a sub-module may designate a selected implementation as the default, leaving it up to the designers of compound modules to select other implementations for special purposes. During simulation, for example, only suspicious components need to be simulated to full detail. Other components may just be simulated functionally at a high level of abstraction. Designers will want to build a specific configuration of a design object for different design tasks and store them in the DDM database alongside the actual design objects.

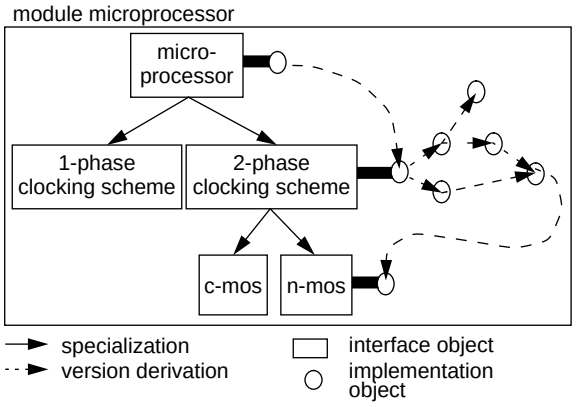

Figure 1. *Inheritance tree of compatible interfaces.*

To summarize, we have the following requirements on the management of design information:

1. Manage design objects from different domains and different levels of abstraction
2. Support abstraction by distinguishing interfaces and implementations of modules
3. Support top-down design. Design objects may be incorporated into a composition hierarchy when only their interface but no implementation is defined yet
4. Support configurations of design objects for different design tasks

3. Information modelling

In this section we describe a conceptual schema that satisfies the requirements stated above. We proceed in two phases. The first phase presents a schema that deliberately abstracts from existing design management schemas to cover our requirements in the most natural way. It is nevertheless necessary to carefully map this schema to the schema of the framework used as implementation platform to provide interoperability with tools integrated into this framework. The second phase maps this schema onto the Nelsis CAD Framework. Both schemas are defined using the Xplain semantic data model [12]. When drawing Xplain graphs, object types are represented as rectangles. *Aggregation* is depicted as a compound type drawn above its attribute types, with lines connecting adjacent *sides*. Roles may be assigned to attributes of a type. *Generalization* is depicted by a line connecting adjacent *corners* of the participating types, the more general type being drawn below the specialization.

3.1 Phase 1: A schema for design management

Figure 2 shows a schema for design management that naturally supports our VHDL-based design methodology. As the
design procedes, a module specification (e.g. "dp32") is stepwise refined; its interface evolves through the abstrac-

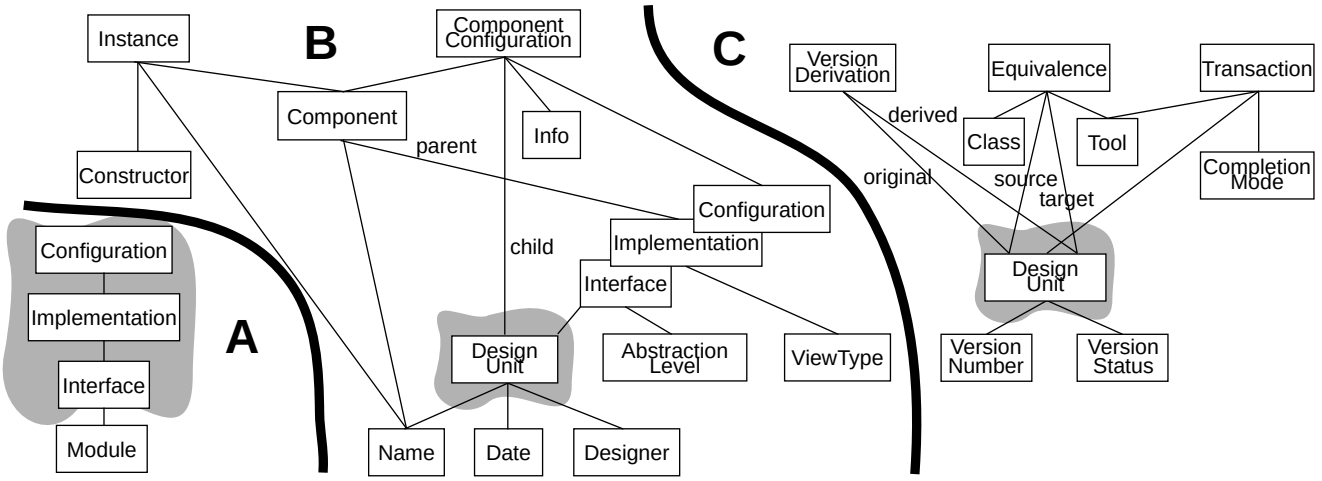

Figure 2. *Conceptual schema for design management. For ease of layout, the single schema is split into three parts. Sub-schema (A) models the inheritance hierarchy of interfaces. Sub-schema (B) represents aspects related to composition hierarchies and configurations. Sub-schema (C) is concerned with dynamic aspects of a design. Design units are marked with a cloud to make them more visible.*

tion levels (e.g. “dp32.system”), each having a number of implementations in different view types (e.g. dp32.system.structure”).

Hierarchy relationships are established between a compound implementation and an instantiated sub-module. The sub-module is represented in the compound module by an object of type *Component*. A component may be instantiated any number of times, each instance being identified by its name and a constructor that specifies the exact circumstances under which it is used in its parent.

Components do not reference the sub-system directly but via component configurations. Component configurations bind a component to a child design unit (either of interface, implementation, or configuration) for a particular purpose. A configuration collects component configurations and may be used as the representative of a design object for a specific purpose like “Release 2", “Fast-CPU”, or “Simulate-ALU”. Configurations can be nested by having component configurations bound to subordinate configurations. Note that our schema does not allow to configure individual instances as is possible in VHDL. This has to be simulated by introducing additional components.

3.2 Mapping to the Nelsis CAD Framework

In this section, we show how to map the design management schema onto the schema used in the Nelsis CAD Framework. Figure 3 gives the overall picture. The fundamental object type in the Nelsis design data management schema is the *DesignObject*. The framework manages design representation at the granularity of this object type. A design object represents an element in a single-view-type-version-set, called *Module*. Every design object is uniquely defined by its version number and its module. Additional attributes hold its version status (either *backup*, *actual*, *working*, or *derived*), and its modification date. A module is uniquely determined by its name and view type. Design objects may be related by the many-to-many relationships *Hierarchy*, *Equivalence*, and *VersionDerivation*. *Transaction* records design transactions performed on a particular design objects and thus allows to retrieve its history. A more thorough explanation of the bells and whistles in the Nelsis schema can be found in [15].

The central problem in mapping the design management schema is how to map the specializations of *DesignUnit*. Our solution is to combine an implementation with its associated interface and represent this pair as a *DesignObject*. This approach ensures that interface and implementation evolve together and can be kept consistent. Configurations are not represented explicitly in the schema, but are emulated by the *install* operation provided by Nelsis and a special flag *BindingMode* on *Hierarchy*. When a dynamic hierarchy relationship needs to be established, a design object with empty implementation part is created and used as the child design object in the hierarchy. In addition, the hierarchy's binding mode is set to *dynamic*.

Whenever a design hierarchy is used by a tool that needs a static binding (e.g. a simulator), hierarchies with binding mode *dynamic* are treated specially. The *install* operation is used to replace the child design object with empty implementation part with a design object from the same module, i.e. one with a “compatible”, possibly refined interface. The *actual* version is chosen if one exists, otherwise the design object with the highest version number will be used.

A module represents a set of implementations with similar interfaces (Figure 4). Implementations with incompatible interfaces are collected in separate modules, related to each other by equivalence relationships.

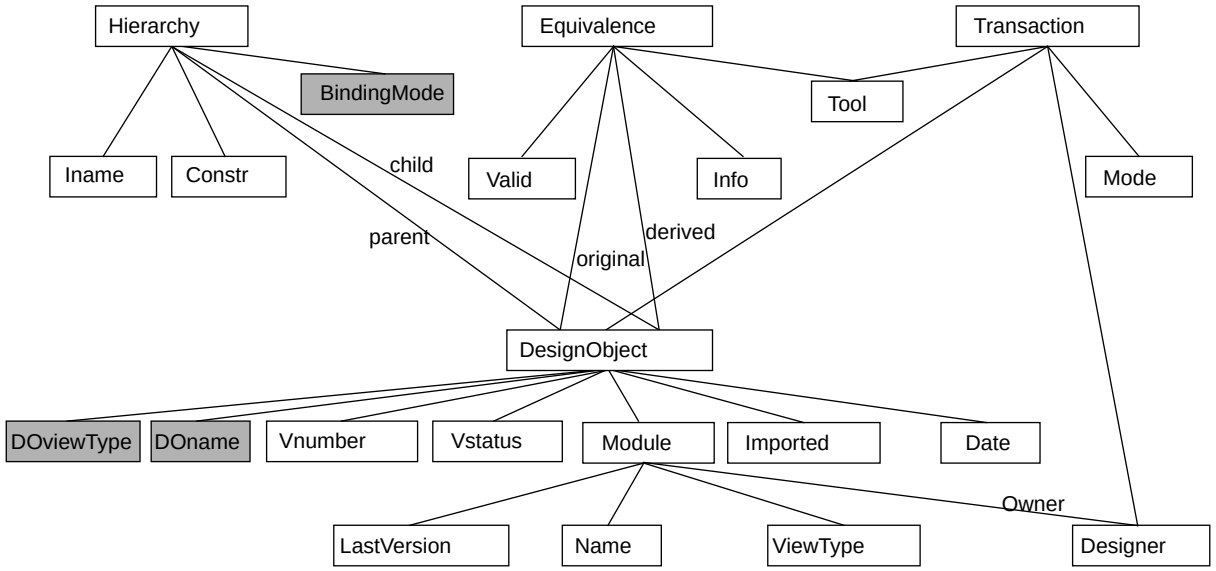

Figure 3. *Mapping to Nelsis. The shaded boxes denote the added types. DOviewType is used to store the view type of an architecture_body (either of "data_flow", "structural", or "behavioural"). DOname stores the name of an implementation. BindingMode (either "dynamic" or "static") marks a hierarchical relation (cf. explanation in the text).*

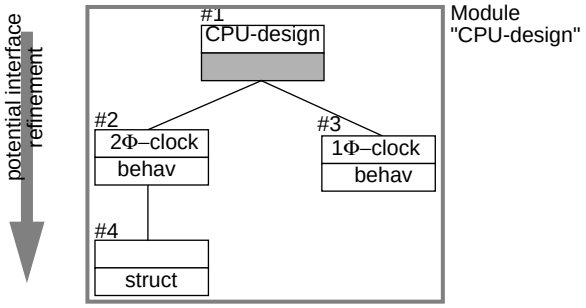

Figure 4. *Implementations with similar interfaces represented by a module. The box at the root of the tree has no implementation and is the target for dynamic hierarchy relationships. Arrows denote version derivation. The figures are consecutive version numbers.*

4. Processing textual design descriptions

One of the foremost features of a framework-based, integrated design environment is to offer design management functions to the designer. The information model presented in section 3 constitutes the conceptual basis to integrate design tools and to organize the various design objects created and manipulated during a design project. Following the assumption that commercial design tools interface to design descriptions stored in files, this section presents the steps necessary to manipulate files with design descriptions in the context of framework-based design management.

Our approach is to extract structural information and design object representation from design files on import and to reconstruct valid design files on export from the design management system. Using this approach, it is possible to exploit the features of a design management system while at the same time being able to use commercial, file based tools:

- a single environment manages all information relevant to a design project, helping to maintain consistency and to foster reuse
- design history is automatically maintained by establishing version derivation and equivalence relationships between design objects
- graphical browsers can guide designers to assess the state and structure of their design

4.1 Design file import

These are the steps to import a set of design files into the design management system:

1. Split the files into text chunks by looking for start and end markers
2. Extract chunk types (e.g. entity declaration, architecture body) and names
3. If necessary, recursively do a thorough syntax analysis of selected text chunks (e.g configuration declarations)
4. Create objects in the design management database and associate them with the corresponding text chunks.
5. Build composition hierarchies from information (i.e. component declarations) in the design files.
6. Establish equivalence relationships between original design objects and their derivations.
7. Resolve configuration declarations into configurations and component configurations on design management.

We accomplish these steps with the aid of parser modules and some extension language scripts. The parser modules are generated from a specification of lexical properties and syntax of the design description language to be supported.

Language specifications can be extremely simple because only little information is needed from a design description. The top-level specification for VHDL looks like this:

```
syntax vhdl {
 token comment -ignore     -pattern <"--".*$>
 token ID                  -pattern <[_a-zA-Z0-9]+>
 token USE                 -pattern <"use"[^;]*";">
 token LIBRARY             -pattern <"library"[^;]*";">
 token END                 -pattern <"end"[^;]*";">

 rule design_file { repeat { toplevel } }

 rule toplevel { LIBRARY | USE
     | "entity" Name:ID guts END
     | "architecture" Name:ID ID EntityName:ID guts END
     | "package" guts END
     | configuration }
 rule configuration -syntax { "configuration" guts END }
 rule guts { list { item | ID } }
 rule item { "procedure" guts END        | "function" guts END
     | "units" guts END                  | "record" guts END
     | "block" guts END                  | "generate" guts END
     | "process" guts END                | "case" guts END
     | "if" guts END                     | "loop" guts END
     | "component" Name:ID guts END      | "for" guts END
     | USE }
}
```

This specification has little to do with the VHDL language specification. It only defines start and end markers as well as names for text chunks. A parser module is generated from such a specification with the aid of YACC and LEX that automatically builds a parse tree of its input. The tree nodes are marked with the start and end offsets of their text regions. By mapping the design files into virtual memory, these offsets can be used directly to locate the text regions and store them as chunks in the design management database.

Certain chunk types need to be analysed to a finer granularity. The corresponding rules are marked with the flag *syntax* (e.g. on *configuration*) to signal the generator that it should generate semantic parse actions to recursively invoke a parser generated from the syntax with the same name as the flagged rule. The parse tree created by this subordinate parser is inserted as sub-tree into the top-level tree, the start and end offsets on its nodes pointing to the same memory mapped design file as its parent.

With this "delegation" of the fine-grained analysis of certain chunk types to subordinate parsers, the top-level language specification can be greatly simplified. Our whole VHDL specification, including the fine-grained specification of configuration declarations, is way below 100 lines, a moderate size compared with the original size of the VHDL grammar.

We use Tcl/Tk [11] as the extension language in our prototype implementation. The kernel language is extended with bindings to design management functions and with two traversal methods on parse tree nodes. As the tree is built during the language parsing by automatically generated code it contains much detail that is irrelevant to information extraction. The traversal methods therefore are defined so that they allow to only look at "interesting" nodes:

```
rule traversals {
       node:IDENTIFIER "all" list { qualifier }
           "-var" var:IDENTIFIER code_block
     | node:IDENTIFIER "one" list { qualifier } }
rule qualifier {
       "-type" type:IDENTIFIER
     | "-tag" tag:IDENTIFIER" }
```

The first variant executes a Tcl code block for every node, optionally considering only those with selected type and tag in the tree rooted at *node*. The node currently looked at is available in the variable named *var* within the code block. The code block may contain *break* and *continue* statements to break out of the traversal completely rsp. to continue with a sibling of the current node. The second variant traverses the tree rooted at *node* and breaks at the first node of type *type* or fails if no such node exists.

With the help of these two methods tree, traversals can be defined that extract the structural information from the analysed design files and create the necessary objects and relationships in the design management database through bindings to the design programming interface.

4.2 Design file export

Export of this object graph back into valid VHDL syntax proceeds in two steps:

1. The DDM database is traversed and a parse tree is created
2. A pre-order traversal is performed on this parse tree, emitting the text chunks associated with each tree node

To create the parse tree, each schema type is associated with a piece of Tcl code that constructs a valid parse tree node for objects of this type. This code block is free to take the text chunk and attributes associated with its object to create a tree node.

4.3 Handling binary files

The import and export mechanisms assume that design files have a well-defined structure and contain some kind of printable description of the design. Quite often, however, design tools produce and expect some kind of opaque, intermediate design description. Examples for this kind of file are results from VHDL analysers, simulation results, or schematics in undocumented, proprietary formats. When no specification of the lexical and syntactical structure of such opaque descriptions is available to the tool integrator, the files containing these descriptions can only be manipulated as a whole. If such files can be associated with a specific design object in a composition hierar-

chy, their contents can be associated with this object and will be imported and exported together with it. If they cannot be associated with a specific design object they have to be attached to the root object of the composition hierarchy and imported and exported whenever a sub-module of this root object is imported or exported.

5. Related work

The research we describe in this paper consists of two main parts. The first part presents a schema for design data management flexible enough to represent structural information extracted from VHDL files and describes how to map this schema to an existing EDA framework. The second part presents a new framework service for design file processing, based on language specification and extension language scripts.

Most of the design management facilities found in EDA frameworks today are based on early work by Batory and Kim, Biliris, and Katz [1,2,10]. This work considers dynamic version binding to various degrees. As this research focuses on database-like functionality, no language processing capabilities are considered.

Ferrans reports on the HyperWeb system for software engineering [8]. While the goal of this system is similar to ours, namely to represent structural information explicitly in an integrated system and to provide powerful browsers and query mechanisms to access and manipulate it, he assumes that the user extracts structural information manually from imported files.

6. A Prototype Design Environment

The following design tools are encapsulated in our current prototype implementation:

- *Import.* Process a set of VHDL files and store them as separate structural and representational information in the framework's database.
- *Text editor.* Text chunks may be viewed or edited.
- *Synopsys design compiler.* When using the design compiler the designer is requested to first create and save intermediate binary files (*.db). A secondary step creates a VHDL file from this binary and re-imports it into the database. No equivalence relationships are currently established between the original (behavioural) design and the synthesized (structural) ones.
- *Synopsys VHDL analyzer and debugger.* After a (hierarchical) design has been validated by the VHDL analyzer it can be debugged interactively.
- *Export.* Export takes a (hierarchical) design from the database and exports it to the file system as a set of VHDL files.

7. Conclusions

We have reported on a new framework service for design tool encapsulation, based on an information model for design management. While several EDA frameworks and software development environments have been reported in the literature that are based on schemas comparable to ours, none offers a general service for tool encapsulation like the one presented. The new service uses generated language processors that perform import and export of design files to and from a design management database. Our prototype design environment is based on the Nelsis CAD Framework and several tools from the Synopsys high-level synthesis and simulation tool suite. The new service shows that the added value of being able to use consistency checks, versioning and versatile browsers offered by an EDA framework can be exploited even for encapsulated design tools.

References

1. Batory, D.S. Kim,W., *"Modeling Concepts for VLSI CAD Objects"*, ACM Transactions on Database Systems, vol. 10 #3, 1985, pp. 322-346
2. Biliris, A., *"Database support for evolving design objects"*, 26th ACM/IEEE Design Automation Conference, 1989
3. Bredenfeld, Ansgar, *"Definition of Modeling Concepts for a Procedural Interface between VLSI-Design Tools and a Common Database"*, Proc. of the 2nd International Workshop on Electronic Design Automation Frameworks, Charlotteville, Virginia, 1990
4. Brielmann, Maria; Kupitz, Elisabeth, *"Representing the Hardware Design Process by a Common Data Schema"*, ACM/IEEE European Design Automation Conference, 1992
5. Carter, Donald E.; Stilwell Baker, Barbara, *"Concurrent Engineering: The Production Development Environment for the 1990s"*, Mentor Graphics Corporation, 1991
6. CAD Framework Initiative, Architecture Working Group, *"Framework Architecture Reference"*, Version 1.2, 1993
7. CAD Framework Initiative, Architecture TC, *"CAD Framework - Users, Goals, and Objectives"*, Version 0.92, 1990
8. Ferrans, James, C.; Hurst, David W.; Sennet, Michael A.; Covnet, Burton M.; Ji, Wenguang; Kajka, Peter; Ouyang, Wei, *"HyperWeb: A Framework for Hypermedia-Based Environments"*, 5th ACM SigSoft Symposium on SDEs, Tysons Corner, Virginia, 1992, pp. 1-10
9. Gajski, D., Kuhn, R. H., *"Guest Editors Introduction - New VLSI Tools"*, IEEE Computer, vol. 16 #2, 1983, pp. 14-17
10. Katz, Randy H.; Anwarrudin, M.; Chang, E., *"A Version Server for Computer-Aided Design Data"*, 23rd ACM/IEEE Design Automation Conference, 1986, pp. 27-33
11. Ousterhout90, John K., *"Tcl: An Embeddable Command Language"*, Winter USENIX Conference, 1990
12. ter Bekke, Johan H., *"Semantic Data Modeling"*, Prentice Hall International (UK), 1992
13. Institute of Electrical and Electronic Engineers, *"IEEE Standard VHDL - Language Reference Manual"*, IEEE Std 1076-1987, March 31, 1988
14. Wagner, F.; Golendziner, L.; Lacombe, J.; de Lima, A., *"Design Version Management in the STAR Framework"*, in T. Rhyne (ed.), Electronic Design Automation Frameworks, North Holland, 1992, pp. 85-97
15. van der Wolf, Pieter, *"Architecture of an Open and Efficient CAD Framework"*, PhD Dissertation, TU Delft, 1993