# Encapsulating Tools into an EDA Framework

**Proefschrift**

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie
aangewezen door het College van Dekanen
op woensdag 13 september 1995 te 13.30 uur

door

*Olav Schettler*
Diplom-Informatiker

geboren te Hamburg, Duitsland

Dit proefschrift is goedgekeurd door de promotor, Prof. Dr. W. Gerhardt

Published and distributed by:

Universiteitsdrukkerij TU Delft
Kanaalweg 2b
NL-2628 EB Delft
The Netherlands

# Contents

**Stellingen**
behorende bij het proefschrift
**Encapsulating Tools into an EDA Framework**
door
Olav Schettler

1. Geïntegreerde Ontwerpomgevingen moeten zich evolutionair in de werksomgeving van designers invoegen. Hierbij zouden bij iedere stap de onmiddellijke voordelen de extra moeite rechtvaardigen.

2. Precieze informatiemodellering en het vroegtijdig gebruik van prototypen zijn noodzakelijk voor de succesvolle vervaardiging van complexe, voordelige, en uitbreidbare informatiesystemen van iedere aard.

3. Het gebruik van nieuwste en complexe technologie is voor het succes van een produkt minder belangrijk dan de continue dialoog met de toekomstige gebruiker, duidelijke en eenvoudige concepten en een efficiënte realisatie.

4. Het openleggen van interfaces en het maken van referentie implementaties voor uitbreidingen door de gebruikers is succesvoller dan de verkoop van een “black box”.

5. Complexe software systemen zullen primitieve operaties gereedhouden, die de gebruiker aan de hand van zijn behoefden flexibel kan samenstellen.

6. “Ondergrondse projecten” zijn vaak een beslissende motor van vooruitgang.

**Stellingen**
behorende bij het proefschrift
**Encapsulating Tools into an EDA Framework**
door
Olav Schettler

1. Integrated design environments have to be applicable by designers in an evolutionary manner. With each step, the immediate improvement in productivity should compensate for the additional effort.

2. Precise information modelling and the early creation of working software prototypes are essential for the construction of complex, useful, and extensible information systems of any kind.

3. The use of complex and latest technology is less decisive for the success of a product than the continuous dialogue with prospective users, clean and simple concepts and an efficient implementation.

4. The publication of interfaces and the creation of reference implementations for user extensions promise more success than the selling of a "black box".

5. Complex software systems should provide primitive operations which can be flexibly combined by users according to their requirements.

6. "Underground projects" often prove as important procreator of progress.

# 1. Introduction

## 1.1 Overview

Today, there is a growing gap between IC fabrication technology and the capability of Electronic Design Automation (EDA) Systems to help design electronic systems. Many design tasks on gate and register-transfer levels can be accomplished by standard tools offered by the big EDA system vendors. On higher levels, however, tool development is still very vivid. The largest productivity improvements are to be expected for design on algorithmic and architectural levels. Such "advanced" solutions often cannot be bought off-the-shelf from the big EDA system vendors but have to be bought from innovative niche vendors or even have to be home-grown.

Innovative electronic design requires close cooperation between system engineers, system architects and IC designers. Design environments with powerful facilities for design information management and browsing are needed to close the communication gap between these people [DeMan 92]. In this thesis we describe how to build an integrated environment for automated electronic design, based on the following observations:

1. Integrated design environments are constructed from framework components and design tools. While design tools perform the actual design steps, i.e. they transform design descriptions, framework components provide an operating environment for design tools and a working environment for designers.
2. The design environment should be open to combine the best available design tools for each design task. This implies that the design tools used have no built-in knowledge about the other tools and framework components in the design environment. They have to rely on standards to accept control input and to transfer design descriptions to and from the other constituents of the design environment.
3. Although it is attractive to use procedural interfaces to interface design tools to each other and to the framework components, none of the proposed solutions is mature enough or generally accepted by tool vendors as the primary design data interface of their choice to use with their tools. For example, the CFI Design Representation Programming Interface [CFI-DRPI 93] is sometimes supported in addition to a tool's native design file interface. No exam-

ple is known to the author where such a standard programming interface is offered as the sole design data interface for a production design tool.

4. Instead, production design tools use (standard) hardware description languages as their primary interface to design descriptions. They transfer input and output data in design files. A design file may contain batches of arbitrary numbers of "design units".
5. Design units are the natural building blocks of an electronic design. Sometimes they represent actual physical devices that are used to assemble a design, sometimes they are logical abstractions that only exist to introduce structure and thus make a design manageable by a team of designers. When designers work on a design, they manipulate design units and their relationships to each other. Design units appear in schematic or block diagrams, are used in part lists, are combined to form more complex design units, and evolve over time. The habit to store and transport design units in files is imposed by the file system centred user view offered by contemporary operating systems.

Traditionally, research on design tool integration has focused on providing procedural interfaces to tightly integrate design tools into a design environment [Hunzelmann 92]. Unfortunately, little of this work has found its way to everyday design reality. Instead, state-of-the-art design tools interface to the outside world by means of standard design description languages like VHDL or Verilog. The obvious solution to integrate file-based design tools with framework components is to have the framework manage design files. This approach is called *encapsulation* and implies that the actual design tool is encapsulated in a wrapper that performs framework-specific house-keeping like setting up an execution environment, importing and exporting design files from a protected database managed by the framework, and communicating with the framework user interface.

While everybody does it, file-based encapsulation is generally considered to be a bad thing for the following reasons:

1. There is considerable overhead in having each tool parse and analyse a textual design description. This holds even more for complex languages like VHDL.
2. Simple encapsulation assumes that all necessary design files can be provided to a design tool before its start. All results produced by a tool are determined

by the way it is invoked. Interactive tools are different in that they allow a designer to manipulate additional design files during the tool run.

3. Designers do not work on their designs in terms of design files but rather in terms of design units and their relationships. When managing complete files, a framework can scarcely help the designer in keeping track of relationships between design units. There seem to be two completely different spheres: (a) tightly integrated design tools, working on individual design units, and (b) encapsulated tools working on files. Sophisticated graphical browsers and query interfaces help designers to keep track of design units and their relationships. Nothing but directory browsers are usable on design files.

Our goal is to provide a new framework service for design tool encapsulation that bridges the mismatch between designers' notions of design units and the design files manipulated by design tools. Our solution is to analyse a design file on input into the framework database and to split it into a set of text chunks each describing an individual design unit. On export, these text chunks may be recombined in flexible ways to form a complete design file which can be fed into a design tool. Thus, the framework can manage individual design units, establish relationships between them and recombine them any way the designer chooses.

We do not attempt to overcome the overhead of design file parsing. Design units are still transported to and from design tools using files as carriers. However, the designer has finer and more flexible control over the selection of design units to be fed to a design tool. Design files are dynamically created from a selected set of design units to be manipulated during a particular design tool run. Hence, the design files fed to a design tool need only contain the necessary design units and thus can be smaller.

We have chosen the following approach to design tool encapsulation:

1. To define the architecture of a new framework service and integrate it into a commonly accepted reference architecture [CFI-FAR 93]. The new service facilitates the mapping between design units and their relationships carried in design files and corresponding objects and relationships managed by the framework. It does so by performing design file analysis and construction. It further starts and controls design tools, communicates with them, and interprets their outputs. The proposed framework service is constructed around a scripting engine [Ousterhout 90] to provide end-user programmability.

2. To formally define a conceptual schema for objects and relationships managed by the framework. Existing framework schemas are either not formally defined [Wagner 91], or do not provide the required flexibility [Kathöfer 92], [Dimes 93a]. Defining a completely new schema has the advantage of being a clean concept and being portable across different frameworks.
3. To provide a toolkit for processing design files and mapping design units found therein to objects managed by the framework. Rather than hand-crafting language processors for important design description languages like VHDL and Verilog, we are of the opinion that a general toolkit provides more flexibility to the environment builder. Thus, the framework is not restricted to supporting a limited set of languages. Support for different languages or dialects of existing languages can easily be constructed.

We describe a prototype design environment that was implemented using this approach. The portability of the conceptual schema was demonstrated by implementing it on a CAD Framework as well as on an object-oriented database. A set of VHDL-based, state-of-the-art synthesis design tools was encapsulated using automatically generated language processors and a small number of scripts.

## 1.2 Design complexity

Electronic designs are complex. Several management techniques have been implemented in integrated design environments over the years to reduce this inherent complexity. Some techniques are unique to electronic design, others were borrowed from software engineering and related disciplines. We assume the following property of electronic designs:

**Property:** All aspects of an electronic design pertinent to its construction, test, and application are formally defined in a set of *design descriptions*.

### 1.2.1 Views

One approach to tackle complexity is to organize the various aspects of an evolving electronic design into the categories *domain* and *level of detail*. Collectively, these two categories are often referred to as the *view dimension* of electronic design [vdHamer 94]. Three, occasionally four domains are distinguished for electronic design descriptions (Figure 1, Walker and Thomas [Thomas 83], [Walker 85], based on earlier work by Gajski and Kuhn [Gajski 83]). The four domains are characterized by the primitives used in design descriptions:

- The primitives in the *behavioural domain* are operators specifying the behaviour of a design. The operators are aggregated into data flow and control flow graphs. Operators may themselves be composed of data and control flow graphs composed of more primitive operators.
- The primitives in the *structural domain* are functional blocks with well-defined interfaces that link to other blocks through networks of electrical connections. Functional blocks may be composed of more primitive blocks.
- The primitives in the *physical domain* are objects with properties like material, area, colour, texture, etc. They are related by spatial relationships like *above*, *below*, *aside*. Physical objects may also be composed of more primitive ones.
- Occasionally, the *test domain* is also considered. The primitives in this domain are stimuli and their associated results. In the sequel, we will concentrate on the other three domains and largely ignore the test domain.

The distinction between domains and levels of detail leads to the following

**Definition:** A *design object* describes an electronic design in a particular domain and on a particular level of detail, as it exists at a particular point in time. The set of all design objects for a single electronic design is referred to as a *module*. A design object is uniquely identified by the tuple (module, domain, level of detail, version).

Synthesis steps transform design objects from a low level of detail in the behavioural domain to higher levels of detail in the structural or physical domains (Figure 1). Synthesis steps are one-to-many transformations, e.g. there are a number of possible sets of ALUs, MUXs, registers, etc. (structural domain) implementing a set of register transfers (behavioural domain). Exploring the design space means to construct a compound design object from any of a number of alternative sets of design objects from the behavioural, structural, or physical domains. This implies that although a design object is associated with a specific domain, it may itself be composed of design objects from different domains. Quite commonly, alternative decompositions of a design object are used for different domains and different levels of detail.

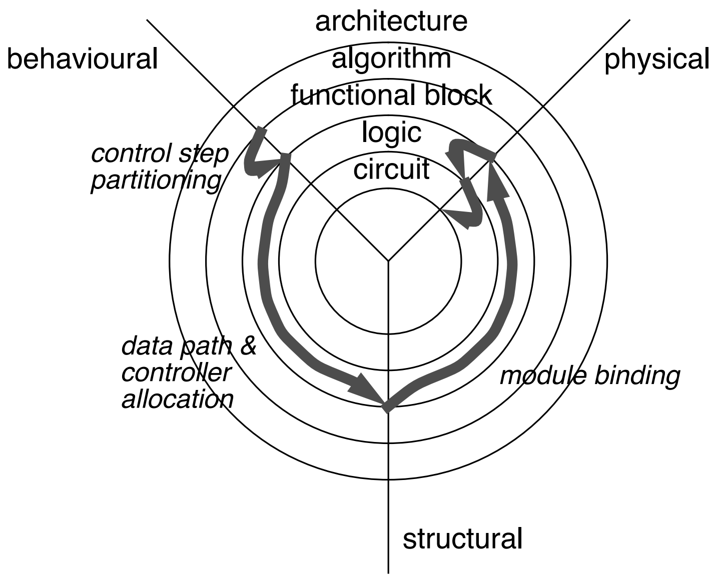


***Figure 1.*** *The Y-Chart by Walker and Thomas, based on work by Gajski and Kuhn. Every design description has a domain (behavioural, structural, or physical) and a level of detail*[*]*. Synthesis design steps are depicted by trajectories taken from one of the outer circles (abstract - low detail) towards the centre (specific - high detail). The diagram shows an example synthesis trajectory from the design methodology implemented in the CMU design automation system. Analysis steps (extraction, simulation) derive an abstract description from a more specific one.*

[*]. This is frequently called “level of abstraction”. To avoid confusion, however, we will rather use “level of detail” in the sequel.

### 1.2.2 Abstraction

Another approach to handle complexity is *abstraction*, accomplished by the distinction of interface and implementation. This particular notion of abstraction was made popular by structured programming in software engineering. The objective for abstraction is information hiding and simplification of reuse. When using a module, the designer should only have to deal with its well-defined interfaces but not with the intricacies of an underlying implementation. It must even be possible to select a different implementation for a given interface, e.g. one on a higher level of detail.

**Definition:** The set of design objects defined for a module is partitioned into disjoint subsets, referred to as *interfaces*. Each design object is placed into exactly one of these interface sets and is associated with exactly one *implementation*. Whereas the interface specifies a design object as seen from the outside, its implementation defines its hierarchical decomposition.

An electronic module 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 world. The behaviour on a very low level of detail (i.e. on architectural level) is reflected in a core interface. The more refined the specification becomes on higher levels of detail, the more the data types for interface elements are refined [Wagner 91]. Table 1 gives some typical data types for interface elements on different levels of detail in the behavioural domain.

| Level of detail | Typical data types for interface elements |
|---|---|
| architecture | Ethernet, RS-232, SBus |
| algorithm | integer, float, array, record |
| functional block | bit, bus, bundle |
| logic | bit_7 |
| transistors | 0...3Volt |

***Table 1.*** *Typical data types for interface elements on different levels of detail in the behavioural domain*

In addition to data type refinement, there may be completely new interface elements on higher levels of detail:

- alternative clocking schemes like single as opposed to 2-phase overlapping clocks
- different power line requirements due to different technologies (+5V and low-power)

| Domain | Interface primitives |
|---|---|
| behavioural | variables |
| structural | signal ports |
| physical | geometries |

***Table 2.** Interface elements in different domains*

Moving to a different domain, however, will almost certainly change the interface, simply because the interface primitives are different (Table 2). Especially design objects in the physical domain have interfaces that are non-isomorphic with those in the behavioural and structural domains due to the radically different kind of information represented in this domain.

Interfaces in one module are related to each other in an inheritance tree each node of which is a refinement of its parent (Figure 2). It may add or refine interface elements defined by its parent. Arranging interfaces in inheritance trees ensures that interfaces in a module are compatible to each other in the sense that refined interfaces can only add new interface elements or refine existing ones but can not introduce in-

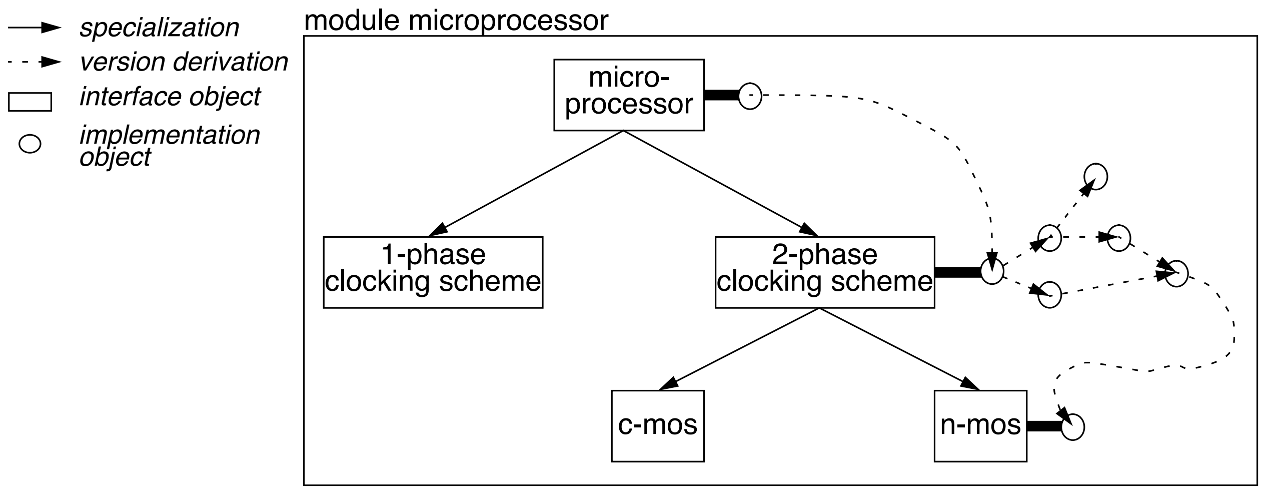


***Figure 2.** Inheritance tree of compatible interfaces. All interfaces belong to the same module and thus are functionally equivalent. Interfaces are linked to implementations which in turn are related by version derivation relationships.*

compatible changes such as deleting interface elements or introducing incompatible type changes. An interface may be associated with a graph of implementations, each of which defines an exact realization of behaviour, structure, or geometry.

### 1.2.3 Hierarchy

The third approach to handle complexity is *hierarchical decomposition*. In the behavioural domain, the implementation of a procedure may recursively contain procedure calls. In the structural and physical domains, the implementation of a compound module may contain sub-modules.

**Definition:** The implementation of a module may make use of other modules. We call such a module a *compound* module; the used modules are referred to as *components*. A component may appear more than once in the implementation of a compound module. The individual manifestations are called *instances*.

There are two important observations related to hierarchical composition:

**Property:** A module used as component typically is itself a compound module, constructed from instances of more primitive modules. Instances two or more levels down a composition hierarchy are sometimes called *occurrences*. It is important to keep track of occurrences to completely specify the hierarchical decomposition of a module.

**Property:** A design object may be hierarchically decomposed into design objects on different levels of detail. It is important to note that the decompositions of design objects in a single module may be non-isomorphic, i.e. there is no one-to-one mapping between the decomposition structures.

## 1.3 Design methodology

We assume a design methodology that proceeds basically top-down. Views, abstraction and hierarchical decomposition are used to reduce design complexity. Design starts with a behavioural description of a design on a low level of detail. 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 is either self-contained or has an interface defined in terms of high-level data types. The design implementation is then partitioned into a hierarchy of functional blocks, each of which is again defined by its behaviour.

**Property:** In general, interfaces are defined *before* their implementations.

First, the interface of each functional block is defined and only then are the individual block implementations realized. The high-level behavioural design description is then stepwise transformed into a physical description on circuit level. A number of commercial and research synthesis tools exist to automatically perform these transformations on functional block level, logic and circuit levels [Walker 91]. Apart from synthesis tools, there are extraction and simulation tools that validate design results and allow to compare the behaviour of implementations in different domains and on different levels of detail (cf. for example [Greiner 93]).

While exploring design alternatives for a complex module, the selection of a specific implementation of a component needs to be flexible. On the other hand, once a module is released to a customer or other developers, its composition must be well-defined and fixed. We use configurations to provide this flexibility.

**Definition:** A *configuration* is associated with an implementation of a particular module. It defines a tree of design objects and subordinate configurations by recursively selecting interfaces, implementations or configurations of components used in this implementation.

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 detail and from different domains may be more appropriate. Or, the designer of a module used as component in other modules 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 questionable components need to be simulated to full detail. Other components may just be simulated behaviourally at a low level of detail. Designers will want to build a specific configuration of a design object for different design tasks and store them in the framework database alongside the actual design objects.

## 1.4 Design description languages

In 1987, the hardware description language VHDL was standardized by IEEE [VHDL 87] and has become a driving force in EDA interoperability today ([Carlson 92], [Heusinger 93]). All major EDA tool vendors support VHDL descriptions as input for synthesis and simulation tools and as result format on various levels of detail in the behavioural and structural domains. Apart from VHDL, there are other mature languages like EDIF [EDIF 88] that are in constant use in electronic design practice, especially for representing circuit schematics and netlists. Yet other formats like CIF of GDSII are used for representing design in the physical domain.

On the other hand, after five years in existence, the Design Representation Programming Interface [CFI-DRPI 93] defined by the CAD Framework Initiative (CFI), the dominant standardization body in the field, is still restricted to representing electrical connectivity data. Electrical connectivity deals with structural aspects of a design on functional block, logic, and circuit levels. As can be seen in the Y-Chart (Figure 1 on page 8), it does not cover all aspects necessary to describe a complete electronic design. To conclude, it is to be expected that language-based design tool interoperability will be common-place for some more years.

## 1.5 Design environments

Successful design of a complex electronic system in a team of designers not only requires the reliable management of design objects themselves but also the maintenance of a complex web of structural information (relationships and annotations) *about* the design objects. Relationships between design objects are used to build composition hierarchies, to associate derived design versions with their originals, or to relate design objects that are functionally or otherwise equivalent. Several researchers ([Batory 85], [Katz 86], [Biliris 89], [Bredenfeld 90], [Wagner 91], [Brielmann 92]) have pointed to the most important relationships: *hierarchical composition*, *version derivation*, and *equivalence* between design objects. In addition, design objects may be annotated with attributes like *level of detail*, *domain*, *owner*, or *status* ([Gajski 83], [vanderWolf 93]).

**Definition:** Design objects may have annotations and relationships which are collectively referred to as *structural information*. The actual design descriptions define the behaviour, structure, and physical properties of a design and are referred to as *representation information*. When the distinction is not important, we collectively talk about *design information*.

This definition is in line with the terms “representational details” and “structural details” as used by Katz et. al. in [Katz 86]. Structural information sometimes is called “meta-data”, representation information is referred to as “raw design data” [vanderWolf 90]. Structural information is accessed by designers in an ad-hoc, query-by-value manner to understand the disposition and history of their design. It is mostly independent of domain and level of detail. Representational information, on the other hand, is accessed by designers and design tools in a navigational manner to get detailed knowledge about the behaviour, structure, and physical properties of a design.

A design environment should serve as an electronic note book to the design team, forming the information basis for more formal forms of project documentation like project binders and manuals. It maintains design information in a component dedicated to the management of design information. The task of this component is to keep structural information accessible, up-to-date and consistent with regard to representation information. Both parts of design information have to be managed in a protected environment to assure that it correctly reflects the state of the design, even in a multi-user environment. A framework component for data management provides basic services like locking, concurrent access, transactions, and object versioning.

**Definition:** The *design information management (DIM)* framework component consists of design information and data management services.

The CAD Framework Initiative suggests the architecture shown in Figure 3 as the reference architecture for a design environment [CFI-FAR 93]. According to this architecture, a design environment consists of a framework, domain independent tools and design tools. To build an environment complying with this structure, design tools have to be integrated with the framework components.

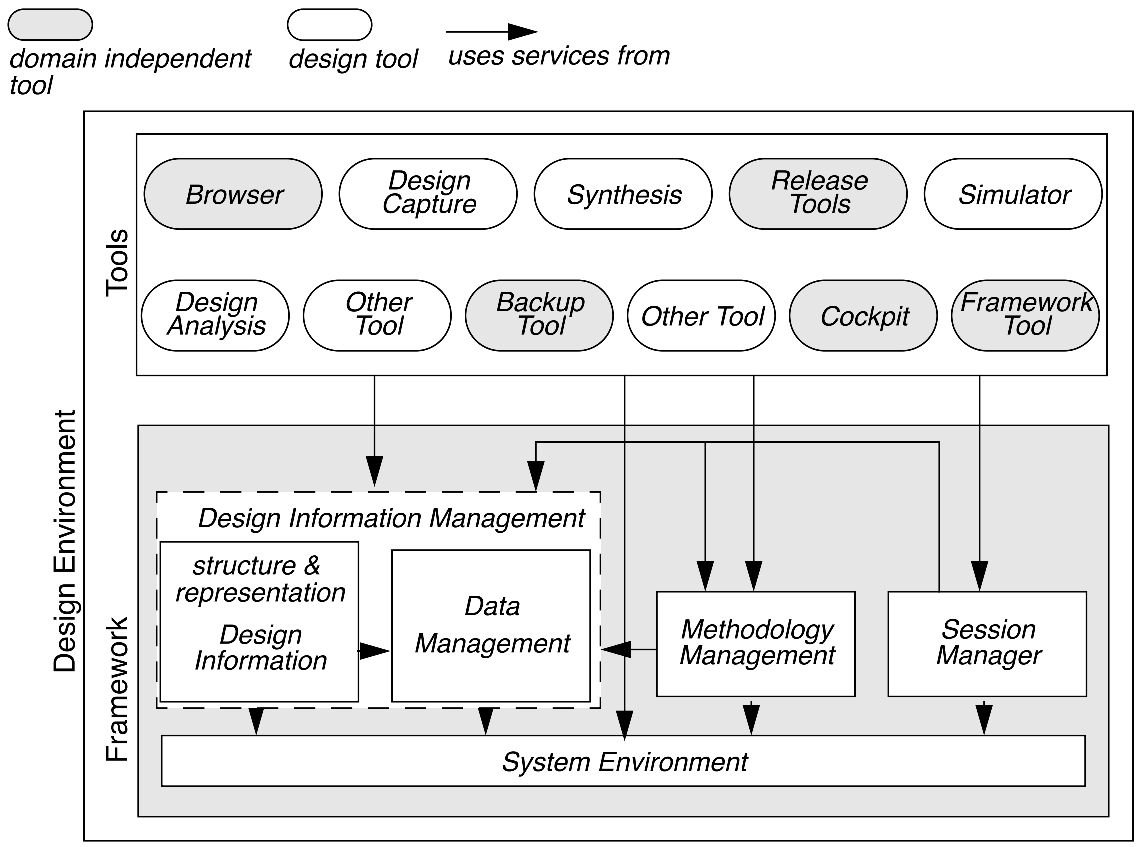


***Figure 3.*** *CFI reference architecture for a framework-based design environment. This high-level view shows the dependencies between framework services and tools.*

## 1.6 Requirements and problems

To summarize, in the area of electronic design we have to meet the following requirements on the management of design information, on design methodologies, and on design environments:

**R1** Manage structural information about design objects

**R2** Manage design objects from different domains and different levels of detail

**R3** Support abstraction by distinguishing interfaces and implementations

**R4** Support top-down design. Design objects may be incorporated into the composition hierarchy when only their interface but no implementation has yet been defined.

**R5** Support configurations of design objects for different design tasks

**R6** Support design description languages

**R7** Support the integration of commercial design tools with framework components

Whereas requirements **R1**-**R5** are a matter of an appropriate conceptual schema for design information, there is a trade-off between requirement **R1** and **R6**/**R7**. The designer thinks of his design in terms of design objects. Design objects have a well-defined interface, they can be versioned, used as components in compound modules, and they can be checked for functional equivalence. Framework tools support this point of view by providing browsers and query interfaces that let the designer browse the design information according to different criteria. On the other hand, language based design tools expect design files as input and produce files as output. In general, a design file contains more than one design object, embedded in some contextual information.

A design project could adhere to the policy that design files describe only single design objects. This way, design files could serve as representatives for design objects and can be used as end-points of relationships and holder of attributes. Whereas input files can be purposefully arranged by a designer to describe only single design objects, the contents of result files are under the control of design tools and beyond the control of the designer. If there is no support from the framework, the designer has to manually analyse the contents of result files, split them at design unit borders and insert them as single files into the data management component. In addition, relationships among result design objects and between derived design objects and the originals already in the database have to be established manually. This is the necessary mode of operation in the JESSI Common Framework [JCF 94a].

In this thesis, we will describe a new framework service that helps to encapsulate design tools to avoid this tedious and error-prone manual processing. Its design is based on the three principles we will develop in Chapter 2. Following these principles, design tools are *encapsulated* into the framework. No access to tool source code is required. With the assumption that commercial design tools interface to design descriptions by means of file input and output, we need to extract structural information and representational information from design files on import into the design information management system and to reconstruct valid design files on export from this system. Using this approach, it is possible to use the browsing and querying facilities of existing framework tools, regardless of whether design objects have been created by tightly integrated tools or by encapsulated, file-based tools. In Chapter 7 we discuss a prototype design environment based on the new integration service that supports the VHDL language. The prototype has been implemented both on the Nelsis CAD Framework and on the object-oriented database management system Object-Store and encapsulates some design tools from the Synopsys suite of synthesis tools [Schettler 94a].

# 2. Design tool integration

## 2.1 Overview

In this section we will review classification schemes used to rate the degree of tool integration achieved in a particular design environment. Unfortunately, the diversity of possible viewpoints taken by researchers and the importance of a good rating for the marketing of an integrated design environment resulted in a muddle of different classification schemes. Classification schemes can be discussed from two points of view, the environment user's and the environment builder's:

> *The environment user is concerned with perceived integration at the environment's interface. [...] The environment builder, who assembles and integrates tools, is concerned with the feasibility and effort needed to achieve this perceived integration.*
> *[Thomas 92], p. 32*

Bad integration[*] from an environment user's point of view results in higher costs during the actual design using the environment. Bad integration from an environment builder's point of view results in high costs for environment construction. This distinction is important because it enables an environment builder to make a set of tools appear well integrated to a user by providing some "glue" between the tools, even if it is not well integrated from his point of view. Thus, although the tools were not built to work in a particular design environment, this glue between tools and between tools and framework services can help to make a design environment better integrated at least from the user's point of view. Of course, it would be more elegant to have all the components of a design environment well integrated in the first place. However, this requires mature, flexible, and standard interfaces to all components in an integrated design environment. Defining such interfaces is a tedious process and due to changing requirements and technology will not be achieved in the near future, if ever.

The goal of using an *integrated* design environment as opposed to a set of independent design tools is enhanced design productivity. In Sections 2.2 and 2.3 we introduce two notions which are key to enhanced productivity in an integrated design environment, namely tool *inter-operability* and tool *inter-changeability.* Tools inter-operate in a multi-user design environment when they can freely communicate and share *at least* control information and data with other tools and framework services.

[*]. according to any of the classification schemes introduced in this chapter

A tool should be easily inter-changeable with another tool, perhaps from another vendor, to supply the same function. This way a design environment can be upgraded to offer state-of-the-art tools for a design task at hand.

The well-established distinction between the orthogonal dimensions and levels of tool integration are discussed in Sections 2.4 and 2.5. Two subsequent sections are devoted to the more controversial aspects, granularity of control and observation, and the black-to-white scale of integration mechanisms. Based on these taxonomies, in Section 2.8 we examine the level of tool observability and controllability that can be achieved by different approaches to tool encapsulation. A synopsis of related work and our conclusions lead to three principles that will serve as the basis of the tool encapsulation methodology to be introduced in the subsequent chapters.

## 2.2 Interoperability

There are four mechanisms of design data handling allowing to supplement or completely replace conventional operating system services by framework technology:

- *Storage* is concerned with persistent archiving of design data and efficient retrieval of this data.
- *Transport* is concerned with the movement of design data between environment components, which may be design tools or framework components.
- *Processing* is concerned with the transformation of design data, either as part of a design step or in preparation of a design step.
- *Browsing* is a technology that ...

> *"... allows engineers to explore the high-level structure of their design. [...] Browsers are important in such an environment because they reveal unobserved, or forgotten structure, enhance design re-usability by making previously designed components easy to find, and assist in communication and shared understanding between different members of a design team."*
> *[Gedye 88]*

Early attempts to apply framework technology focused on enhancing *tool-to-framework* inter-operability (e.g., the Oct System [Harrison86]). A common approach was to build a central database with a global schema that allows to store and access all aspects of a design. Design data was then accessed by tools through a programming interface, processed in core, and then written back to the database. Transport is not an issue for database-oriented systems because design data is accessed in

small portions directly via a procedural interface. Browsing of design data is ideally supported because all aspects of the design are randomly accessible.

Much work has been invested in trying to use relational database systems for design data storage and access. As the performance of relational database proved inadequate for the navigation-oriented access patterns of most design tools, special-purpose design data handlers emerged. Most of the major CAD environment vendors today base their integrated environment products on such special purpose design data handlers. Their success probably stems largely from the fact that the integrated tools were directly developed for the particular design data handler in use and so can optimally exploit its features. Lack of standards, unfortunately, has up to now prevented a wider use of (object-oriented) database-oriented systems despite their obvious advantages.

Tool vendors, and those users that depend on the openness of their design environments, still prefer file based design data storage, transport and processing. The emergence of standard design description languages like VHDL has removed the importance of framework supported data handling to achieve good *tool-to-tool* interoperability. Tools can be written in a framework-independent manner, relying solely on fairly stable standard languages as interface specification. Storage and transport of files is well supported by operating system services, so that hardly any support is needed by enhanced framework services. Manually creating design file processors is tedious but can be obviated by the application of efficient parsing techniques provided by widely available compiler construction toolkits.[*]

Unfortunately, with file based design data storage and transport, *browsing* becomes a major hassle. Browsable relationships between design objects are now obscured within files and have to be extracted. The extracted structural information then has to be kept up-to-date with respect to the design representation stored in files. A common conclusion by framework vendors is to condemn file-based design tools

[*]. There is an interesting divergence in the use of languages as opposed to programming interfaces in the UNIX and MS-DOS worlds. Under UNIX, versatile language processing tools have always been available. In addition, the UNIX tool suite is specially oriented towards text processing. The use of text files therefore is common-place under UNIX. By contrast, under MS-DOS language and text processing tools are not available. Also, the fact that all programs share the same address space makes it easy to use public programming interfaces across program boarders. The net result of this fundamental difference in operating system philosophy is that under MS-DOS and its offspring MS-Windows, programming interfaces are versatile and widely accepted by users as well as tool vendors. Under UNIX, text files are the interface metaphor of choice. Only recently, with industry standards like ToolTalk, there is a chance that this situation may change.

altogether and to put off environment builders until database technology is more mature. We want to show in this thesis that with some care even with today's technology effective browsing of design data can be achieved. The important idea is to clearly *distinguish storage from transport mechanisms*. While design tools rely on design data being transported as files, they need not be stored this way. We will discuss this in more detail in Section 2.6 where we take a closer look at granularities.

## 2.3 Interchangeability

As CFI states, ...

> *"... design System builders tend to judge a tool strictly on its merits as a tool, not on its compatibility with a proprietary framework. They want the freedom to select the best tool for their needs. In current systems, the process of selecting a tool is strongly influenced by the proprietary framework in which the tool operates. In many situations, the tool of choice cannot be easily coupled to the Design System currently in use. Users want to divorce decisions about tools from decisions about frameworks. This will result in a more seamless Design System for the user."*
> *[CFI-UGO 90], O2.2, p. 17*

There are two aspects of design tool interchangeability: *Open interfaces* and *design tool abstraction*. The common, standard framework backplane that CFI suggested in 1990 to solve the lack of open interfaces has neither been specified nor realized until now. Rather than waiting for its coming, users and independent tool vendors have moved to standard design description languages to solve their tool integration problems.

> *"Tool abstraction is the process of reducing the specific details of successfully invoking and executing a tool to a common set of parameters understood by the framework and the user. Tool abstraction aids the designer both in selecting the proper tool for a particular function and in correctly invoking the tool without requiring intimate knowledge of a tool's specific invocation syntax or execution environment."*
> *[CFI-UGO 90], O2.3, p. 18*

Furthermore, tool abstraction can be used to register a design tool with a framework so that different framework components and other tools can effectively coordinate with the new tool ([ECMA 91], p. 11). The success of CFI's proposed standard tool abstraction mechanism is documented by the fact that commercial (e.g., ViewLogic) as well as research frameworks (e.g., Nelsis) have adopted CFI's Tool

Encapsulation Specification format as the method of choice for registering new tools in the system.

## 2.4 Integration dimensions

It is commonly agreed that there are orthogonal dimensions to integration. In accordance with Wasserman we identify five dimensions [Wasserman 90]:

- *Framework integration* or its weaker form, *platform integration,* is concerned with the effective use of a platform's services. Its goal is to provide the basic elements on which agreement policies and usage conventions are built. The framework is considered an extension of the operating system platform.
- *Data integration* is concerned with the use of data by tools and framework services. Its goal is to ensure that all the information in the environment is managed as a consistent whole, regardless of how parts of it are operated on and transformed.
- *Control integration* is concerned with communication and inter-operation among tools and between tools and framework services. Its goal is to allow the flexible combination of an environment's functions, according to project preferences and driven by the underlying processes the environment supports.
- *Presentation integration* is concerned with user interaction. Its goal is to improve the efficiency and effectiveness of the users' interaction with the environment in fulfilling his design tasks by avoiding distraction through incompatible visual presentation paradigms.
- *Process integration* is concerned with the role of tasks and tools in the design process. Its goal is to ensure that tools interact efficiently in support of a defined process. A process is *"a specific combination of tools and/or other processes that performs a design function"* [Fiduk 90]

If we interpret the integration dimensions as spatial axes spanning an n-dimensional cube, we can associate a complete design environment, clusters of well integrated services, or single tools with a point in this space, depending on our focus (Figure 4). The coordinates of such a dot give a measurement of the level of integration of a given item in each of the integration dimensions. As no quantitative measure exists for the level of integration, such a diagram should only be regarded as a rough statement of the level of integration of two or more items relative to each other.

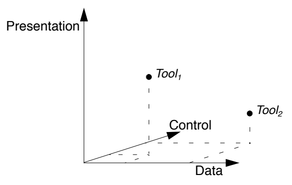


***Figure 4.*** *Dimensions of integration. Each axis represents a dimension. The coordinates of a point in the spanned, n-dimensional "integration space" can be used as measurement for the level of integration in each of the dimensions for the item represented by the dot. For example, $tool_1$ is hardly integrated with respect to control and data, but well integrated with respect to presentation. The integration of $tool_2$ focuses on data and control integration and neglects presentation integration.*

This diagram leads to the conclusion that the integration dimensions are all equally important in constructing a design environment. Emphasizing either of the dimensions would therefore be more a matter of personal taste than dictated by added value to environment users. When looking more closely at this added value, however, it becomes clear that this is a misconception. Although the sum of integration levels for tools 1 and 2 in Figure 4 may be the same, due to its lack of data and control integration the integration of $tool_1$ may be of less value to a designer's productivity than the integration of $tool_2$. In fact, the integration dimensions can be arranged in a sequence according to their importance in achieving an overall high level of integration, ranging from essential to merely desirable:

1. *Framework integration* provides the technical foundation for all other integration dimensions. Thus, without framework integration or its weaker form, platform integration, there is not much point in trying integration in any of the other dimensions. It is so basic that many authors do not regard framework integration as one of the integration dimensions at all but rather as one of the invariants on which all other integration dimensions are based.
2. *Data integration* is the next important integration dimension. Without data integration, data as the output of one design tool has to be converted to the format expected by another tool. Before the advent of standard design description languages like VHDL or EDIF, writing the necessary converters

was the most expensive effort during the construction of an integrated design environment from off-the-shelf design tools and framework components. This effort was often so trying that achieving good data integration was considered the only important problem in tool integration (assuming a common platform), whereas the other integration dimensions were completely neglected.* It was hoped that by using a database management system (DBMS) for design data storage, all data integration problems could be easily solved, but the use of a DBMS merely transferred the need to lexically and syntactically process design descriptions. The need for agreed semantics between the design tools still remains and the design of a new common conceptual schema for each new DBMS-based integration project binds much of the resources available in such a project. As already mentioned above, a major drawback of using a database management system is that it is hard to write stand-alone tools. This is the reason why DBMS-based data integration is mostly used in proprietary systems and is hardly used by independent tool vendors.

3. *Control integration* can be subdivided into *provision* and *use*. Without control integration, automation is not possible because every single activity has to be triggered explicitly by the user. With control integration, however, each component in the environment can be designed to either *notify* other components of relevant events or to *control* their operations. Hence, efficient control integration allows to construct new complex services or tools from existing ones, fostering modular environment construction. For tools to share functionality, they must be able to communicate the operations to be performed. As operations require data, the tools must also communicate data or data references. Control integration therefore relies on the existence of data integration.
4. *Presentation integration* not only involves compliance of the user interfaces of tools with a common appearance and behaviour, it also involves that tools update their user interfaces to always give a consistent view on the current design state. To accomplish this, tools have to be able to receive notifications from framework services, thus relying on control integration.

*. Compare the results of the German DASSY project (Data interchange and interfaces for open, integrated design systems) as documented, for example, in [Hunzelmann 92], a joint effort of German universities and industrial research institutes to define a common tool interface. Despite the project's claim to cover all aspects of tool integration only data integration was considered in depth.

5. *Process integration* relies on both data and control integration. Sequencing of design tasks would not make much sense if tools were unable to exchange design data. Automatically running processes require that tools are able to react to events that reflect their preconditions in a process. Tools should be able to generate events that help to satisfy other tools' preconditions. This requires control integration. Furthermore, presentation integration is important when a running process is to be monitored by a browser tool.

As long as the only interface to state-of-the-art design tools is via design description files, only proprietary systems will be able to exploit the full range of integration dimensions. A tool that internally processes design objects will not be able to communicate its need for additional objects to a data management service that only recognizes files, not the objects hidden in these files. Design browsers will not be able to display relationships between design objects described in the same design file. Design constraints attached to certain design objects will not be able to trigger necessary events when only the modification date of a file changes as the result of an update operation but nothing is known of the changes that were made to the file's content.

All of these deficiencies are due to the fact that design information is hidden away in design files. Although we can expect that we will have to deal with design files for quite a time, much higher levels of presentation and even process integration can be accomplished if the framework components recognized the contents of the design files they handle.

## 2.5 Integration levels

In the last section we frequently mentioned the level of integration in each of the dimensions. Although there is no quantitative measure for these levels, the introduction of a coarse classification scheme can help to compare one specific integration with others. This classification scheme was introduced by Brown and McDermid [Brown 92], who have also defined several integration dimensions that somewhat deviate from the more common terms introduced above. They use

- *tool integration* to refer to a combination of our data and control integration,
- *interface integration* to refer to our presentation integration, and
- *process integration* in accordance with our terminology.

Their dimensions *team integration* and *management integration* have no resemblance in our system. Brown and McDermid apply integration levels only to data and control integration: they remark that obstacles in these dimensions must be removed first before the presentation and process dimensions can be explored in more detail. We extend their notions to presentation integration.

1. *Carrier level integration* is concerned with the basic physical prerequisites of integration. For data integration, this e.g. would mean to agree on using physical files without providing any common module to aid in the processing of data. Every single tool thus is responsible for data structuring, validation, and so on. The basic means to exchange data is in place, but nothing is yet known of the data's structure. In the control integration dimension, carrier level integration e.g. would mean to agree on using Remote Procedure Calls for exchanging control messages but saying nothing about the expected messages or parameter types being passed. In presentation integration, carrier level integration could mean fixing the use of the X11 windowing system without any restriction on the use of a particular widget set.
2. *Lexical level integration* is concerned with lexical conventions. In data integration, this could mean that all tools would use a common preprocessor on data files that resolves macro definitions and include files. Another example would be the convention to use a leading "." in text files to introduce a control command. Nothing is yet said about which control commands would mean what to a particular tool. In control integration, on the lexical level a certain format for control messages would be adhered to. This level of integration is for example achieved by Sun's ToolTalk without taking into account a specific set of message types. In presentation integration, on this level we would fix a certain widget set, e.g. Motif or OpenLook, without defining how these widgets should be arranged in a typical design tool.
3. *Syntactic level integration* is concerned with structuring conventions. On this level, tools agree on a set of data structures like parse trees in data integration, message categories like object creation, deletion, and change notification in control integration, or common application skeletons with a menu bar with "File", "Edit", and "Help" buttons on top, a scrolling window in the middle and a status line at the bottom. This is the level most commonly found in today's integrated design environments.
4. *Semantic level integration* goes a step further and defines a common understanding of data structures' semantics. This information can either be hard-

coded in the integrated tools or can be contained in a data dictionary for ease of reference and modification. With data integration, not only the syntactic structure of a design description has to be agreed on but also the semantics of the data. In control integration, a message dictionary would be set up which defines an agreed upon set of messages to be used. In presentation integration, a set of higher level, interface building blocks like complete graph flow browsers could be offered. Only proprietary, closed systems have reached this level of integration up to today.
5. *Method level integration* determines not only the kinds of objects which tools deal with, the kind of control messages they may exchange and the interface building blocks used in their construction, but fixes the use of the tools themselves within a methodology.

## 2.6 Granularity

The selection of an appropriate granularity is of key importance for design tool integration. Although this certainly holds for all integration dimensions, here we focus on data integration because all higher dimensions can profit from good data integration. As stated above, we may distinguish between the data handling mechanisms storage, transport, processing, and browsing. When looking at contemporary integrated design systems, we can distinguish three different granularities: *File-based, object-based,* and *value-based*.

File-based design data handling is the coarsest granularity found in integrated design systems today. It is dictated by file-based design tools. What is actually required by such tools, though, is file-based design data *transport* to and from the tools. The tools themselves *process* design data at the granularity of individual attribute/value-pairs. *Storage*, when concerned with version derivation, transactions, and equivalence relationships between individual objects, is most naturally done at the granularity of design objects. This is in line with the approach taken in the Nelsis CAD Framework [vanderWolf 90] and with earlier work (e.g. [Katz 87]). At this granularity, the framework stores information *about* the design objects as structural information, but does not need to make any assumptions about the actual detailed design representation. Assuming that tools handle the conversion from files to values in their input/output processors, our task is to easily convert from objects to files and vice versa, to bridge the granularity mismatch between object-based storage and file-based transport.

## 2.7 Integration approaches

When talking about tool integration, commonly only two extremes are distinguished, based on whether a tool's source code has to be modified to achieve integration or not:

**Definition:** *Black-box integration* or *encapsulation* assumes that the tool's source code contains no function calls to framework services. The tool only interacts with the native operating system services. All interaction of the tool with framework services uses a wrapper that translates tool data and control requirements to appropriate calls to framework services.

**Definition:** *White-box integration* or *tight integration* assumes that a tool is integrated into a framework by embedding function calls to framework services in the tool's source code. All interaction with framework services can thus be done directly by the tool.

Sometimes an intermediate integration approach *(grey-box integration)* is also considered [Kathöfer 90]. With this approach, the framework data-handling system emulates a file system and stores design data as unstructured objects. It offers the usual programming interface to tools, with functions like *open, seek, read, write,* and *close*. Like white-box integration, the tool's source code has to be modified to call these functions instead of the operating system's ones. Modifications are much simpler, though, because the file system functionality is closely emulated. While the data-handling system can offer extra protection of the design data by access control and transactions, like black-box integration, it has no knowledge of the internal structure of the design data and therefore cannot aid in management and browsing of design data structure. For this reason, we will not consider grey-box integration in the sequel.

Integration approaches are often associated with granularity of design data transport and storage. As encapsulated tools normally expect design data to be transported in files, both storage and transport are assumed to be file-based. We have reasoned above that file-based design data storage prevents effective management and browsing of design data structure. On the other hand, tightly integrated tools are assumed to transport design data across a programming interface in a value-based manner, therefore storage is often also assumed to be value-based.

Often, a distinction in control granularity is also assumed. Encapsulated tools are started and then left unattended until they either succeed or fail. Tightly integrated tools give access to more fine-grained functionality that may be invoked individually. This distinction may be true in most of the cases where the only way to control the functions a tool performs is through appropriate command line parameters. Nowadays, only few design tools are this uninvolved. Most provide a complex command language that can be used to control their actions. Interfacing to the command interface of a tool is desirable even if there is no programming interface to it, to be able to control the tool more flexibly.

## 2.8 Tool observability and controllability

Design tools manipulate design data by transporting them from a source, processing them, and transporting them back to a target*. Controllability deals with the question

> *"How and to what degree can a* program *invoke atomic operations in a tool."*

Observability deals with the question

> *"How and to what degree can a program tell which design data a tool reads and writes, and in what status it leaves them?"*.

It is important for integration that both kinds of interaction with a tool can be performed by programs and not just through the tool's user interface. The more observable and controllable a design tool is, the more tightly it can be integrated into a design environment. A tool is written to interact with a particular execution environment. For tools designed with a particular set of framework services (e.g. design data; data, methodology, and session manager) in mind, the execution environment comprises of the programming interfaces of these framework services. Most standalone tools, however, only rely on operating system services (e.g. file system, processes, terminals, window system). Here the execution environment comprises of the system calls offered by these operating systems services.

*. We have used the vague terms "source" and "target" here to account for the abundance of different ways to build the interface between a design tool and its design data, ranging from files managed by the operating system, streams passed between the tool and a framework or other tool through a socket or pipe, or even calls to a programming interface.

We now introduce two encapsulation approaches, a-priori encapsulation and tracing encapsulation, which differ in the amount of observability and controllability they assume of encapsulated design tools:

**Definition:** With *a-priori encapsulation*, a tool wrapper anticipates any possible request for an environment service by the tool in advance and prepares the execution environment accordingly.

The JESSI Common Framework, for example, ([Kathöfer 92], [JCF 94a]) assumes the following, fixed execution sequence to perform a design step:

1. The framework checks out a complete design hierarchy into the file system.
2. The framework invokes a wrapper shell-script, passing it environment variables and the name of a design object as command line parameters.
3. The wrapper preprocesses and copies the design files to appropriate places, sets additional environment variables, assembles a command line and invokes a design tool with it.
4. The design tool performs the actual design step, resulting in either success or failure.
5. The wrapper checks this result to decide on a number of post-processing alternatives, again processes and moves files and finally informs the framework of success or failure of the design step.
6. The framework in turn checks in the resulting design files and decides on their status depending on success or failure of the design step.

More flexible solutions may scan a set of working directories for new files or examine log files produced by the tool to deduce the actions taken by the tool.

**Definition:** With *tracing encapsulation*, a tool wrapper traces requests for environment services by any of a number of possible ways, logging notifications and servicing requests.

If the tool natively interacts with an operating system, the tool wrapper has to emulate operating system service requests issued by the tool (e.g. 'open design file X') and to translate them on the fly into the appropriate framework services calls. There are a number of ways to achieve this:

- In operating systems that support shared libraries, the *open* system call may be replaced with a routine that knows about framework services and redirects file

requests to it. Design tools do not have to be recompiled but automatically use the new *open* routine when at start-up they dynamically load the system library containing it.
- Where this is not feasible, the *ptrace* system call allows to divert control at system calls to the wrapper at the cost of some execution time overhead.
- An altogether different approach does not try to manipulate the tool but rather lets the framework data-handling system emulate an NFS file system. The workstation at which the tool is running mounts this file system as usual and from then on design tools would see no difference between an ordinary file system and the one managed by the framework. The framework, on the other hand, has complete control over the design data read or written by the tool. It has to provide directory entries according to the design objects it manages. Whenever a tool opens one such object for read or write, the framework emulates file accesses on this object.*
- A last resort is to either trace log files created by the tool or regularly scan working directories touched by the tool to find out what happens.

A major problem with tracing encapsulation is that it is very tool-specific and can require much effort to realize. We can distinguish four cases of interaction of a tool and its execution environment (Table 3):

| | | environment | |
|---|---|---|---|
| | | passive | active |
| tool | passive | **1** | **2** |
| | active | **3** | **4** |

***Table 3.*** *Cases of interaction between a tool and its execution environment*

1. With a passive environment, all tool/environment interaction has to be anticipated before the actual tool run and the execution environment has to be set

*. While this approach seems to require much programming effort, the NFS protocol is publicly defined as internet standard [NFS], and the public domain operating system *Linux* contains an NFS server as starting point for an implementation [Linux].

up accordingly. Tool activities may be logged but not influenced. The classical way of encapsulating a design tool using a shell script can only result in this kind of interaction. Even with this low level of interaction impressive results can be achieved [Casotto 90].
2. An active environment intercepts environment interactions by the tools and can react accordingly. With this kind of interaction it is for example possible to check out additional design objects as files into the file system, start servers on demand, inhibit certain operations, or request user interaction.
3. An active tool requires that either the tool is designed from the start to issue logging calls to its environment or can be modified to do so. The programming interface of Casotto's VOV system, e.g. only requires the four functions *VOVbegin, VOVend, VOVinput,* and *VOVoutput* to be inserted into the tool's code.
4. The highest form of interaction can be achieved if both tool and environment are active. This way, the tool can actively issue requests to be serviced by the environment. A wrapper can translate operating system oriented requests into ones understood by the framework.

The problem with approaches (1) and (3) is that *all the control is on the side of the framework*. Once the wrapper is started, no more design objects may be requested from the tool side. Also, *all* result files have to be determined in advance, making it necessary to split generic design tools with complex input/output relations into myriads of 'activities', one for each combination of input/output files.

If approaches (2) and (4) are to be combined with wrapper encapsulation, the standard UNIX shell as scripting language has to be replaced by a language more tailored to the task of tool encapsulation. We refer to the wrapper scripts as *tool execution protocols* to emphasize the fact that these scripts define a communication protocol between the designer, design tools, framework services, and the operating system. An execution protocol is comparable to a handshake protocol used in networking to define the state transitions of communicating agents. Figure 5 on page 37 shows an example of the communicating objects or 'agents' that send and receive messages in a simple execution protocol.

Each of the agents implements a number of methods:

| Agent | Component | Method | Description |
|---|---|---|---|
| Designer | | selectDO | use any of the means offered by the framework to select a design object |
| | | select Activity | select an activity to apply to the design object |
| Frame-work | Framework tool | offerDO | offer a design object for manipulation |
| | | offer Activity | offer an activity depending on the design object type |
| | | notify ActEnds | present the result of the activity |
| | Design information manager | check-OutDO | check out a design object (flat/hierarchical) into the |
| | | grantDO | notify the availability of a design object |
| | | checkInDO | check in a design object from the file system |
| | Session manager | startTool | start a tool or connect to running tool |
| | | startActivity | start an activity within a tool |
| Operating System | Process Manager | | start/stop/signal tools |
| | File System | | store files hierarchically |

***Table 4.*** *Methods implemented by communicating agents*

| Agent | Component | Method | Description |
|---|---|---|---|
| Design Tool | Activity | requestDO | ask for a designated design object |
| | | logActivity | write log about the activity: execution time, execution environment, affected design objects |
| | | accessDO | read a design object extracting its contained design representation |
| | | activityEnds | signal success/failure of the activity |

***Table 4.*** *Methods implemented by communicating agents*

Table 5 on page 36 depicts a sample execution protocol, written as messages exchanged between agents. The framework offers the designer a set of design objects through its user interface. After selecting, one out of a set of activities (methods of the selected object's type) becomes available. The framework finds the design tool associated with the selected activity, starts it if necessary and sends it the request for the activity, passing the identifier of the selected design object as parameter. The tool requests the denoted design object from the framework, passing a desired placement in the file system. When the checked-out design object is granted and placed in the file system the tool starts the requested activity, requesting additional design objects when needed. On completion the framework is notified of success/failure. The framework checks in the results and notifies the designer of activity completion. In this execution protocol, once an activity is started, it is in control to access design objects needed to complete its task. With this mode of operation, the design information manager has to provide access to design information and automatically keep track of objects in use to protect them against tool failure and conflicting concurrent access.

| | agent | recipient | message | parameters |
|---|---|---|---|---|
| 1. | f/w[a] | designer | offerDO | set<DO[b]> |
| 2. | designer | f/w | selectDO | DO |
| 3. | f/w | designer | offerActivity | set<activity> |
| 4. | designer | f/w | selectActivity | activity |
| 5. | f/w | o/s[c] | startTool | tool-path |
| 6. | f/w | tool | startActivity | activity options DO |
| 7. | tool | f/w | requestDO | DO placement flat/hierarchy |
| 8. | f/w | o/s | checkOutDO | placement |
| 9. | f/w | tool | grantDO | DO placement |
| 10. | tool | f/w | logActivity | DO activity |
| 11. | tool | o/s | open/read/write/close | DO path |
| 12. | tool | f/w | activityEnds | DO activity |
| 13. | f/w | o/s | checkInDO | placement |
| 14. | f/w | user | notifyActivityEnds | DO activity |

***Table 5.*** *A sample execution protocol*

a. framework
b. design object
c. operating system

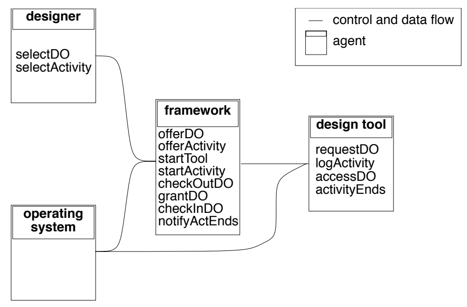


***Figure 5.*** *Agents in a simple design system*

## 2.9 Related work

In this section, we will take a look at the literature on tool integration. It is widely recognized now that there are distinct dimensions to tool integration. Our dimensions *framework integration, data integration, control integration,* and *presentation integration* are based on work by Wasserman, who also developed the coordinate space diagram [Wasserman 90]. Thomas and Nejmeh refine Wasserman's original work with a focus on data integration [Thomas 92]. They identify five properties of data integration between the data management/representation aspects of two tools: *interoperability, nonredundancy, data consistency, data exchange,* and *synchronization* according to the following table:

| persistent data | non-persistent data |
|---|---|
| interoperability | data exchange |
| data consistency | synchronization |
| nonredundancy | |

***Table 6.*** *Properties of data integration*

Two tools are well integrated with respect to interoperability and data-exchange if they require little work to be able to use each other's data. The distinction of persistent and non-persistent data is made because the mechanisms that support data-exchange integration may be different from the mechanisms that support interoperability integration. Two tools are well integrated with respect to data consistency and synchronization, if each tool indicates its actions and the effects on its data to other tools that might by affected. Again, the distinction of persistent and non-persistent data is made to account for different mechanisms. Two tools are said to be well integrated with respect to nonredundancy if they have little duplicate data or data that can be automatically derived from other data. Nonredundancy applies to both persistent and non-persistent data. The coverage of different dimensions for good integration is now also requested by standardization bodies ([ECMA 91], [CFI-FAR 93]). They focus, however, on the "core" dimensions data, control, and presentation integration.

Brown and McDermid add the dimensions *team integration* and *management integration* [Brown 92]. The prime contribution of their work, however, is their introduction of integration levels, which they discuss in the context of data and control integration with a focus on data integration. The justification for this restriction is the fact that "most IPSE's[*] have only tool integration[†], and even there the integration potential is underused and the state of the technology is unsatisfactory." ([Brown 92], p. 25). Of course, this statement applies to software engineering.

Granularity is an important issue in recent work on tool integration. Researchers with a strong background in databases associate different levels with granularity, file-based granularity being the "lowest" and value-based or "fine-grained" granularity being the "highest" level [Kathöfer 90]. We rather agree with the more designer-oriented point of view that neither of these extremes make much sense for the end user of a design environment. The designer expects support by a framework and framework tools in the management of structural information, which is preferably managed at an object-based granularity [vanderWolf 90]. This view-point seems to be well supported by work done by Katz et al. at Berkeley, looking at both design data management and suitable user interface metaphors for design systems ([Katz 87], [Gedye 88], [Silva 93]). We know of no work that distinguishes between the granularities for storage, transport, and processing.

[*]. IPSE: integrated project support environment

[†]. tool integration: our data+control integration with a strong focus on data integration

The terms *encapsulation* and *tight integration* are common knowledge among researchers in the ECAD domain (e.g. [CFI-UGO 90]). A problematic aspect of this distinction is that it is often associated with integration granularity [Kathöfer 90], even more so because it is automatically assumed that file-based design data transport implies file-based storage. With tool encapsulation, there is some disagreement as to the amount of knowledge that should be provided about a tool in its tool abstraction specification. CFI captures only information relevant for tool invocation (tool name; executable pathname; argument type, name, and default values; return status; expected input and output data) [CFI-TES 93]. In the ULYSSES II System, Parikh et. al. require some more information to be able to select a tool suitable for a design task based on its abstraction [Parikh 93].

Tool control and observation is simple if a tool is tightly integrated into a framework and performs all the interaction with its environment through programming interfaces of framework services [Kupitz 92]. If a tool is not designed to operate in a framework environment and its source code is available, it can be enhanced to call functions from a small programming interface to mark actions relevant to design management [Casotto 90]. If, however, no source code is available, any of the encapsulation techniques described above has to be applied.

## 2.10 Conclusions

In the introduction we noted that the object-based management of structural design information separate from design representation information supports the designer's view of his design data. On the other hand, design tools clearly process design data on the granularity of individual values. Finally, transport to and from encapsulated design tools is mostly accomplished on the granularity of files. Value-based or *fine-grained* design data handling has often been advocated as a solution to this mismatch. We feel, however, that - from a design management point of view - framework supported, fine-grained design data handling is not feasible and, in fact, not desirable for the following three reasons:

- Whereas design tools are centred around a specific and detailed understanding of the design objects they manipulate, a framework needs only a coarse-grained, object-based perspective biased towards structural information to provide execution contexts for its integrated tools. 'Coarse-grained' in this case is not to be confused with 'file-based'.

- As already pointed out in Section 1.5 on page 13, design structure tends to be domain independent information accessed by designers in ad-hoc, value-based queries. Design representation, on the other hand, is accessed by design tools by traversing the graph of related design objects, starting from some named or top-level design object.
- Design tools are written to an up-to-date model of the design objects they handle. In fact, all too often a particular design language dialect is exclusively defined by the sole tool that works on it. A framework is always second to come and will rarely have exactly the same model used by the tools it serves. Keeping the model up-to-date is expensive, and hardly worth the effort.

To conclude, we base our tool integration approach on the following

## Three principles:

1. Design structure information is managed at the granularity of design objects.
2. Design representation information is transferred from and to design tools in design files.
3. For the purpose of this thesis, we assume design tools to be encapsulated.

Principle 1 guarantees that design data are managed (stored, browsed, and queried) at a granularity that is natural to designers and optimally matches the requirements for meaningful relationships like hierarchical composition, version derivation, and equivalence. Principle 2 matches the most popular way in which design tools interface with the outside world. This and principle 3 ensure that the source code of design tools need not be available to successfully integrate a design tool into an EDA environment.

# 3. Workman's tools

## 3.1 Information modelling

In this thesis, we present a technical system that solves problems in the world of electronic design. We have chosen to base its design on an information model.

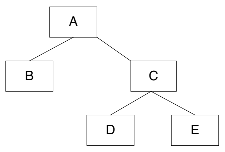
**Definition:** An *information model* defines the relevant *concepts* in a particular application domain in terms of their *attributes* and *relationships*. The synonym *conceptual model* is often used for information model.

In terms of database technology, an information model abstracts from the fine-grained detail of an actual implementation. To be of any value, the information model must be based on some well-defined formalism.

**Definition:** A *data model* defines a type system, consisting of a number of base types and constructors to derive new types from existing ones. It also defines a data access and manipulation language that operates on type extents and a language to specify integrity constraints.

Early attempts to define data models stem from the area of database design. The first data models (hierarchical and network data models) directly described the physical data organization supported by a database management system (DBMS) and provided only low-level operations for data manipulations. The relational model is based on the formally defined relational theory and effectively insulates the user of a DBMS from the physical data organization. It provides, however, only scalar datatypes (boolean, integer, real, string) and flat relations as datatypes and therefore is not well suited to model technical domains.

A data model that allows to capture more semantics is the entity-relationship (ER) model by Chen [Chen 76]. This model represents information in terms of entities and the relationships between them. A major advantage of the ER model is that a graphical notation is defined for it which is the usual technique to capture ER models. Since its introduction, many extensions have been proposed to make it more suitable for the modelling of technical systems like electronic circuits. Of these, especially aggregation and generalization are of importance ([Smith 77], [Batory 85]).



**type** E: INTEGER.
**type** D: { 'red', 'blue', 'yellow' }.
**type** C = D, E.
**type** B: STRING ('default').
**type** A = B, [C].

***Figure 6.*** *Xplain modelling concepts, graphical notation, and textual schema language. The graphical notation is ordered, i.e. an aggregate is always drawn above its attributes. This way, the edges in a diagram are associated with cardinalities. In the example, there is exactly one element of B for each element of A, while one element of B is related to zero, one, or many elements of A. Furthermore, A is a specialization of C: While C has the attributes D and E, A has the attributes B, D, E.*

With the more widely spread acceptance of the object-oriented concepts and the quest for a richer type system the EXPRESS language, now an ISO proposed standard [Spiby 93], has gained wider acceptance. EXPRESS is an object-oriented language used for information modelling. It comes with a graphical notation (EXPRESS-G) that provides graphical idioms for a subset of the textual EXPRESS language. EXPRESS-G diagrams share a major disadvantage with ER diagrams in that they are unordered. No natural entry point into the diagram can guide the reader in understanding such a diagram. More important, depending on the entry point chosen by a reader, an ER or EXPRESS-G diagram can mean different things.

We have chosen the Xplain semantic data model [terBekke 92] over the other approaches for its few and well-defined concepts and clear graphical notation (Figure 6).

The key concept in Xplain is a type:

**Definition:** A *type* is defined by a unique name and properties. A type is associated with an *extent*. The extent of a *base type* is a set of either integer or real numbers, strings, or named constants from an enumeration. The extent of an *aggregate type* is a set of objects, each consisting of a unique identifier and a value. The value is a tuple, each element being from the extent of one of the base or aggregate types.

In addition, Xplain allows to *specialize* an aggregate type by adding new attributes to an existing type. For example, Table 7 shows the type extents of the types shown in Figure 6.

| type | extent |
|---|---|
| A | $\{ (b, d, e) \mid b \in B, d \in D, e \in E \}$ |
| B | $\{e \in Strings\}$ |
| C | $\{(d,e) \mid d \in D, e \in E \}$ |
| D | {'red', 'blue', 'yellow'} |
| E | $\{e \in RealNumbers\}$ |

***Table 7.*** *Type extents of the example in Figure 6*

Xplain defines two inherent integrity rules for the types in an information model [terBekke 93]:

- *Relatability*
  Each attribute in a textual type definition is related to exactly one type with the same name as the attribute. Every type may correspond to a number of attributes.
- *Convertibility*
  Each type definition is unique; there are no two type definitions carrying the same name or the same set of attributes.

An important property of this model, resulting from the relatability integrity rule, is that a given type can have different interpretations (e.g. attribute, aggregation, or generalization) with regard to related types. We will not explain the data manipulation and constraint definition languages here because there are few and intuitive constructs. Examples used in the sequel should become clear from the associated explanations.

## 3.2 Syntax specification

Frequently in this thesis we have to specify syntax. Although we could have used standard Extended Backus-Naur Form (EBNF) for most specification purposes, we preferred to use the same language for syntax specification that we implemented as support language for encapsulation tasks. As this language is more verbose than the commonly used EBNF notation the syntax specifications should be more easily understood.

The syntax specification language allows to define both lexical properties and syntax in a single specification file. While at this point this is merely a matter of convenience, it will become essential when using the language to capture input and output syntaxes in a single specification. Simple keywords can be directly placed into the grammar as symbols. Token classes have to be defined before their use by means of regular expressions.

### 3.2.1 Lexical properties

A language specification consists of format free text. There are comments, string constants, patterns, identifiers, keywords, and a few special characters. Both kinds of "C++" comments, bracketed in "/*" and "*/", and from "//" to end-of-line, are recognized. String constants are enclosed in double quotes and support the usual "C" escape characters "\n" for line-feed and "\t" for tabulator. Patterns are regular expressions enclosed in angular brackets ("<", ">"). All regular expressions of the scanner generator tool *lex* [Lesk 75] can be used. Identifiers are case sensitive and have the usual "C" appearance <[_A-Za-z][_A-Za-z0-9]*>.

The following keywords are defined:

```
-export, -ignore, -left, -nonassoc, -pattern, -prec, -right, -separator,
-syntax, -type, list, opt, prec, range, repeat, rule, syntax, token
```

These are the token definitions for the specification language:

```
token LINE_COMMENT -ignore <"//" .* $>
token BRACKETED_COMMENT -ignore <"/*" .* "*/">

token ID -pattern <[_A-Za-z] [_A-Za-z0-9]*>
token PATTERN -pattern <"<" [^>]+ ">">
token LITERAL -pattern <\" ( [^"\\] | \\[nt\\"] )* \">
```

### 3.2.2 Constructs for Extended Backus-Naur Form

The specification language supports the following EBNF constructs:

```
rule word {
  | "opt" alternatives
  | "list" separator alternatives
  | "repeat" separator alternatives
}
rule separator {
   opt { "-separator" LITERAL }
}
```

Lists may define a separator literal like "," or ";". All EBNF constructs can be replaced with non-extended constructs.

- Optional parts are enclosed in curly braces and prefixed with the keyword *opt*. The replacement is:
  ```
  rule opt { /**/ | alternatives }
  ```
- Optional lists are enclosed in curly braces and prefixed with the keyword *list*. The replacement for lists without separator is:
  ```
  rule list { /**/ | list alternatives }
  ```
  Lists with separators are resolved as follows:
  ```
  rule sep_list { /**/ | tmp }
  rule tmp { alternatives | tmp separator:LITERAL alternatives }
  ```

- Lists with one or more elements are enclosed in curly braces and prefixed with the keyword *repeat*. The replacement for lists without separator is:
  ```
  rule repeat { alternatives | repeat alternatives }
  ```
  Lists with separators are resolved as follows:
  ```
  rule sep_repeat {
    alternatives | sep_repeat separator:LITERAL alternatives
  }
  ```

### 3.2.3 Structure of a language specification

A language specification consists of one or more named specification modules. Each module defines some comment conventions, lexical tokens, symbol precedence, and grammar rules:[*]

```
syntax meta_spec {
  rule modules {
         repeat { "syntax" ID "{" list { token | prec } repeat { rule } "}" }
  }
} // syntax meta_spec
```

### 3.2.4 Declarations

Declarations are used to define lexical language properties and symbol precedence. Token definitions associate a regular expression with a grammar symbol. The property *-ignore* may be used to declare a token as comment.

```
rule token {
   "token" ID opt { "-ignore" } "-pattern" PATTERN
}
```

Symbol precedence and associativity may be defined by a number of precedence statements, each with a list of literals or grammar symbols. Precedence and associativity are used in the generated parser to resolve grammar ambiguities. The earlier precedence statements list the literals and grammar symbols with low precedence. The later a precedence statement appears, the higher the precedence of its literals and grammar symbols.

[*]. When presenting bits of syntax specification, we will frequently only use single modules or even only a few rules and will then omit the module header for simplicity.

```
rule prec {
   "prec" prec_property list { ID | LITERAL }
}
rule prec_property {
   "-left" | "-right" | "-nonassoc" | /**/
}
```

### 3.2.5 Grammar rules

The main part of a language specification consists of a list of grammar rules. A grammar rule specifies a left-hand side and a list of alternatives as right-hand sides. Each alternative in turn may consist of a list of words.

```
rule rule {
   "rule" ID alternatives
}
rule alternatives {
   "{" list -separator "|" { words } "}"
}
```

A word may either be

- a literal or a pattern,
- an identifier that denotes a token or a rule,
- an optional list of alternative words,
- a list (with either zero or one as the lower element count) of alternatives,
- or a subordinate set of alternatives.

Words may be tagged to give them a unique name for reference. A rule alternative may be assigned a precedence by naming a grammar symbol in the *-prec* property. The alternative receives the same precedence as was assigned to this symbol in a *prec* statement.

```
rule words {
   list { opt { ID ":" } word } opt { "-prec" ID }
}
```

```
rule word {
   PATTERN
 | ID | LITERAL
 | “opt” alternatives
 | “list” separator alternatives
 | “repeat” separator alternatives
 | alternatives
}
rule separator {
   opt { “-separator” LITERAL }
}
```

The syntax specification language as presented in this section will be extended by some specific constructs to enhance its usability as specification language for design file processors in Section 6.4 on page 111. There, we will also look into how efficient design file processors can be generated mostly automatically from such a specification.

# 4. Tool encapsulation architecture

## 4.1 Overview

Principles 1 (“granularity of design objects”) and 2 (“design representation in files”) presented in Section 2.10 clearly deviate from the classical assumption that granularities of design data storage, browsing, and transport are equal. In this scenario, design data are either handled at a file-based granularity or at a value-based granularity. On the other hand, we want to achieve granularities as follows:

| aspect of data handling | storage/ browsing (DIM) | | transport (file system) | | processing (tool) |
|---|---|---|---|---|---|
| granularity | object-based | **(1)** | file-based | **(2)** | value-based |

***Table 8.*** *Granularities for storage, transport, processing and browsing of design data*

While gap (2) is bridged by the input/output processors of design tools themselves, our encapsulation mechanism has to consider gap (1). As outlined in the Introduction, this encompasses

- the analysis of design files and storage of extracted structural information in the design information management component of the framework, and
- the construction of design files from structural information and design representation managed by the framework.

The structural information is managed separately from the design representation information which is stored as text chunks attached to design objects. On export, these text chunks have to be recombined as required by the current design step and design configuration to be fed into the design tool.

For control integration, our minimum requirement is that we have to be able to communicate an initial request for some activity from the framework to a design tool and to transfer notifications of activities on design objects between framework and tool. Despite this low level of control integration without intermediate requests being

passed to the tool, there can be no general solution due to a lack of agreed standards in the area of tool control.

Even less can be achieved for presentation integration. As tools generally do not provide access to their internal object structures, it is by and large not feasible to combine browsers that work on framework data structures as well as on tool data structures. Integration in this area has to be achieved on a case-by-case basis. At least, due to the possibility to manage meaningful relationships between design objects in the design information management (DIM) component, useful browsers can be implemented on top of it.

Principle 3 of Section 2.10 prescribes that design tools be encapsulated into the framework. Yet, with purely file-based storage and transport, writing the appropriate tool wrappers can be quite lengthy and error prone due to the lack of standard behaviour on the side of design tools. The amount of work would simply be overwhelming if we attempted to bridge the gap between object-based storage and file-based transport in an equally ad-hoc manner. A new framework service is called for. This service replaces the UNIX shell conventionally used for programming wrappers for encapsulated tools. By looking again at the execution protocols introduced in Section 2.8, we conclude that such a service must be able to perform the following tasks:

1. Determine affected design objects from tool arguments
2. Elaborate dynamic version bindings
3. Map between physical design files and text chunks attached to design objects in the design information manager
4. Check in/out design objects
5. Establish an execution context (file system directories, environment variables, global libraries, start-up files, resource files)
6. Start tool
7. Maintain message connection with activity in tool
8. Scrutinize tool execution

## 4.2 A new framework service

We assume that tight integration is too expensive to be feasible for a majority of design tools so that in many cases only encapsulation will be attempted. We assist design tool encapsulation by an architecture that directly supports the tasks

addressed by an environment builder with appropriate services. As a first step towards such an architecture, we refine the CFI reference architecture (cf. Figure 3 on page 15) as shown in Figure 7.

In this refined architecture, an encapsulated tool is augmented by a set of tool-specific adaptors. While the tool is not aware of its encapsulation into a framework, these adaptors connect to the public interfaces offered by the framework services in much the same way as tools that are directly integrated with these framework services would. From the framework point of view, an augmented tool behaves just like a tightly integrated tool. As suggested in this architecture, tool adaptors are considered tool specific and not subject to direct framework support. Thus, ad-hoc techniques prosper, giving little room to the reuse of adaptor code.

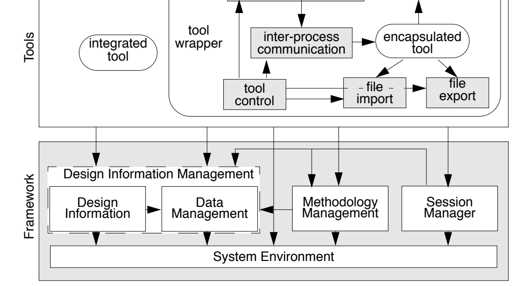


***Figure 7.*** *Service view of the CFI reference architecture. Also shown is an encapsulated design tool with its tool wrapper that performs file import and export and some other services.*

Our approach is to consider the adaptors needed for tool encapsulation as additional framework services. Code can be shared among tools with similar adaptor requirements. For example, all tools that interface to the design description language VHDL can reuse the same file processing adaptors. Encapsulated simulators can share the same inter-process link to a text editor that displays the source line currently executed. We conclude that adaptors for encapsulated tools should be part of the framework (Figure 8). While Figure 8 identifies the tasks addressed by the new tool encapsulation service, we will introduce a detailed component architecture in the next section.

## 4.3 Component architecture

### 4.3.1 Language selection

The new encapsulation service is built as a language driven, programmable engine. This architecture provides the required flexibility to cope with widely different encapsulation tasks. At the heart of this engine is an interpreter of the Tool Command Language *Tcl* [Ousterhout 91].

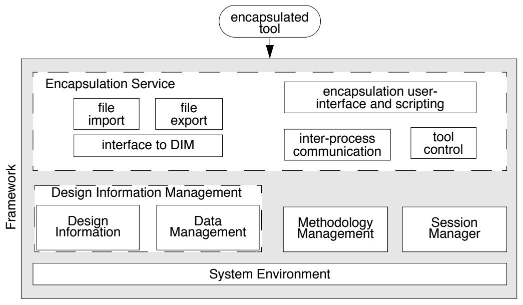


***Figure 8.*** *Service view of the framework, extended with tool encapsulation service*

The main reason to select Tcl over CFI's choice *Scheme* ([CFI-EL 91]) is its modelessness:

> *"Tcl does not enforce any particular programming paradigm such as functional, logic, or object-oriented."*
> *[Brannon 94], p. 12*

The language is not to replace the framework's extension language but to supplement it with a language that is more familiar to tool integrators already being exposed to UNIX shell programming. Tcl was chosen over its competitors for the following reasons:[*]

- The language syntax is close enough to UNIX shell languages so that tool integrators, familiar with writing shell wrappers, will quickly adapt to the new language.
- It is interpreted, so tool integrators and even designers can readily configure the design environment. This is especially useful when encapsulating new design tools or extending the granularity at which design data is managed by the design management system. Not least, it provides an easy-to-use environment for rapid prototyping of new design management tasks.
- It has but one data type, i.e. character strings. Experience shows that what looks like a serious flaw both for script writers and from the standpoint of efficiency, hardly ever is a restriction. However, it dramatically simplifies the interface to the language from "C" as well as the language itself. When more complex types are needed, time-critical parts of an encapsulation wrapper can always be written in "C". The "C" code is demand-loaded into the engine and becomes available as new Tcl commands. From Tcl scripts there is no visible difference except for improved execution speed.
- It has a complete language kernel with variables, lists, arrays, procedures, as well as string and file handling capabilities. An object-oriented extension is available to help structuring large applications.
- It is designed to be embedded in applications. It has an orthogonal, well designed and documented procedural interface that makes it easy to define new commands, access variables and other state information in an interpreter.

[*]. Despite this list of good arguments in favour of Tcl, our new encapsulation framework service could have been based on Scheme, with some more effort needed for the implementation and adaptation to a lisp-like language for the tool encapsulator.

- It is easily extendible. A single procedure call in "C" is required to register a new command with an interpreter.
- It is integrated with a powerful and easy-to-use, Motif compliant toolkit for the X11 Window System as well as for Microsoft Windows, so creating execution protocols which require fancy user interaction is possible not depending on a single platform without too much effort. The key features of this toolkit are a constraint driven geometry management and useful default values for all widget parameters so only the really necessary values have to be given.
- A versatile extension *expect* exists that allows to control a design tool via its input/output.
- Extensions exist which allow to easily use mechanisms for inter-process communication (IPC) such as sockets, remote procedure calls, or ToolTalk [ToolTalk].

### 4.3.2 The Tool Command Language

For the purposes of this thesis we will explain only that part of Tcl's syntax and semantics that is necessary to understand the Tcl code examples given in the sequel. For a more thorough introduction to Tcl and the Tk toolkit the reader is referred to [Ousterhout 94]. A Tcl command consists of a command name and a list of command arguments, separated by white space. Newline characters or semicolons are used as command separators. Each Tcl command returns a string result and a code signalling success or failure.

Four additional syntactic constructs give the language a Lisp-like character:

- Curly braces act as nestable quote characters used to group complex arguments.
- Square brackets are used to invoke command substitution.
- Dollar signs are used for variable substitution.
- Backslashes may be used to insert special characters into arguments.

There is only one data type in Tcl: *Strings*. ASCII strings are used for commands, command arguments, results returned by commands, and variable values. Some commands expect one or more of their arguments to have one of the special forms of string: list, expression, or command. What makes Tcl extremely usable in our

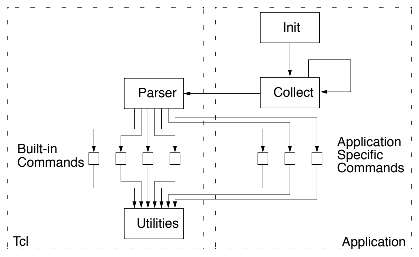


***Figure 9.*** *Structure of an application that uses a Tcl interpreter [Ousterhout 90].*

context is that it is easy to integrate into applications. The interpreter consists of a small library of "C" functions used to invoke Tcl commands, to bind "C" functions defined in the application to new Tcl commands, and to manipulate the values of Tcl variables (Figure 9).

We have extended the interpreter with new commands to fit it to its new task as execution protocol engine. Key to the extension is a module we added to make compiled object-oriented classes, written in "C" or "C++", accessible from Tcl scripts. Objects can thus be created, manipulated and deleted. The public features of compiled classes have to be registered to the Tcl interpreter. This is achieved either by invoking a schema programming interface or by feeding the interpreter a schema via extended Tcl scripts. The wrapper facility interfaces to objects through a small set of well-defined procedures which have to be implemented for each public member of a class. We have implemented a generator that wraps the more regular cases of public data members and member functions, constructors, and destructors of "C++" classes.

It is desirable to load the parsers for particular design description language into the encapsulation engine on demand. Thus, support for new languages can be added or existing modules replaced when requirements change. We have therefore added a facility that is able to load a module with compiled class definitions into a running encapsulation engine. Demand loading can be triggered while loading a schema for a compiled class into the Tcl interpreter.

Other than the generic modules for the wrapping of compiled classes and for demand loading, the encapsulation engine is equipped with modules more specific to the task of tool encapsulation (cf. Figure 10):

- *Language independent parse tree and symbol table*. This module contains classes to construct and manipulate syntax trees and symbol tables in-core. A corresponding schema is fed to the encapsulation engine through the Tcl interpreter to wrap these classes and make them accessible from execution protocols.
- *The mapping between DIM Application Programming Interface (API) and the base framework API* is realized as a Tcl script and is loaded into the encapsulation engine at start-up time.

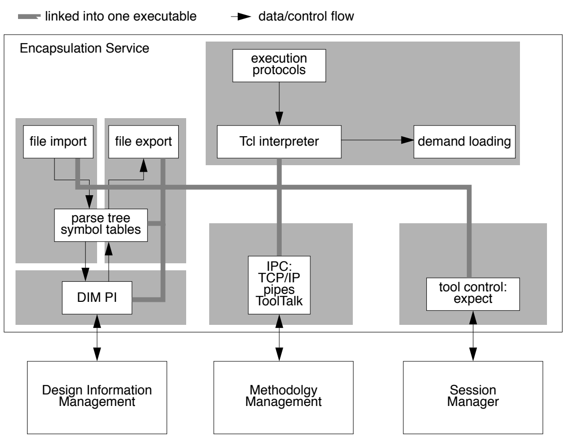


***Figure 10.*** *Component architecture of the encapsulation service (services in Figure 8 are shaded)*

- *File import and export*. Whenever a new language needs to be supported, a new language parser module is generated from a specification and dynamically loaded into the engine. In addition, reading out information from the parse tree and storing it in the DIM system on import is performed in a Tcl script that is specific to a design description language. Export can completely rely on the information stored in DIM and is therefore carried out by standard methods, invariant to design language and tool changes.
- *Execution protocols*. Assuming that language parsers for the supported design description languages exist, the tool integrator is mostly concerned with writing execution protocols for the design tools to be encapsulated. Execution protocols handle all the details of preparing a tool run, starting and controlling the tool, and finally cleaning up and evaluating the results.

## 4.4 Wrapping compiled "C" and "C++" objects

The execution protocol engine needs to make a number of compiled modules accessible to the tool integrator. These include the procedural interface to the underlying DIM component, the scanner and parser for design description languages generated by scanner and parser generator tools, and the implementation of parse trees. Other modules, e.g. to support designer interaction, are conceivable. Although these modules may not be implemented in an object-oriented language like "C++", the object-oriented paradigm is well accepted and so even pure "C" implementations offer opaque handles as object identifiers. We have therefore chosen to uniformly wrap compiled modules with object-oriented classes, regardless of whether the underlying implementation is fully object-oriented or only object-based. A basic requirement for the wrapping mechanism was that it should blend nicely with the object-oriented extension *iTcl* [McLennan 92] we have adopted for our extended Tcl interpreter.

### 4.4.1 Schema representation

The challenge for the wrapping mechanism is to map between object identifiers in the compiled module and strings in the extended Tcl interpreter. As objects may be created both from compiled code and from execution protocols, the strings representing object identifiers need to be dynamically created whenever a new object is encountered. Figure 11 shows the information model we use to manage the schema

of a compiled module. As it is used to represent schema information, it can be regarded as a meta schema for compiled modules.

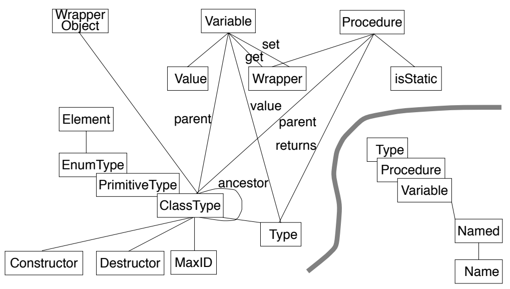


***Figure 11.*** *Schema for the object wrapping module. The name attributes of* Type, Procedure, *and* Variable *are inherited from a common supertype* Named.

Every compiled object we want to handle from an execution protocol is associated with an object of type *WrapperObject*. A *WrapperObject* has a unique *Handle* and identifies its associated compiled object. *Identifiers* can be in-core memory addresses for "C" or "C++" objects, or database identifiers for objects residing in an (object-oriented) database*. *WrapperObjects* are typed, that is, they are associated with a *ClassType* object.

As subtype of type *Type*, *ClassTypes* are named. In addition, they may have a number of data members (objects of type *Variable*) and member functions (objects of type *Procedure*). *ClassTypes* may also be involved in multiple inheritance by maintaining a list of *ancestor-ClassTypes*.

*Variables* and *Procedures* both have a *Name* and (value- or return-) *Type*, which can be either a *ClassType*, *EnumerationType*, or one of the *PrimitiveTypes* void, integer, boolean, float, or string.

*. The types *Handle* and *Identifier* are not depicted in the schema diagram because they have a one-to-one relationship with type *WrapperObject*.

*WrapperObjects* access data members and member functions of their associated wrapped objects via wrapper procedures written in "C". Data members are wrapped by associating them with a pair of wrapper procedures for read and write access. The *set-Wrapper* procedure may be omitted for read-only data members. Two different signatures are defined for these wrapper procedures:

```
typedef int (*Enc_Wrapper)
  (Enc_Info* info, Enc_Identifier obj, Enc_Value& result,
   int parmCnt, Enc_Variable parm[]);
```

Procedures of type *Enc_Wrapper* are used to wrap ordinary member functions or give read or write access to a data member. An object identifier is required.

```
typedef int (*Enc_StaticWrapper)
  (Enc_Info* info, Enc_Value& result, int parmCnt, Enc_Variable parm[]);
```

Procedures of type *Enc_StaticWrapper* are used to wrap static member functions or constructors where no object identifier is needed as a parameter.

Using a purely procedural interface to wrapped objects has the advantage that even objects implemented in "C" may be wrapped. Only some kind of opaque handle mechanism is necessary which is a common concept in both operating system libraries (e.g. FILE*), and design data management systems (e.g. CELL_HANDLE, STREAM_HANDLE in Nelsis).

The implementation of this meta-schema facility provides a set of classes and member functions to easily define *Types* and associate them with *Variables* and *Procedures*. At run-time, each *ClassType* object maintains its extent in a hash-table to efficiently map from handles to object identifiers. The reverse direction is also supported: Whenever a new object identifier is encountered, maybe as a result of a procedure call or as value of a data member, that has not yet been assigned a handle, a new object handle is generated consisting of the class name and a unique number.

As the whole system is not persistent, object handles are only valid within a single session and are not suitable as object identifiers per se. It is, however, possible to override the standard handle generation procedure and use e.g. a persistent object id as handle. This is achieved by writing a new procedure

```
char* genId (Enc_Type* type, void* obj);
```

and registering it with the schema programming interface. Thus, handles may be made valid between sessions or among different instances of the encapsulation service and can be used directly as object identifiers in inter-tool messaging.

Now that we are able to describe types implemented in compiled modules, the next step is to be able to load such modules into a running execution protocol engine. New modules, e.g. for new or modified design description languages, can be added to the engine by the tool integrator on demand. No relinking of the engine is needed. The implementation relies on the UNIX library functions *dlopen*, *dlsym*, and *dlclose* to implement this feature. The only requirement on the demand loaded module is that it must contain position independent code. This can be achieved with a compiler switch.

Using a procedural interface to define a schema has little overhead but is complicated to use and difficult to maintain. We have therefore defined two new Tcl commands that allow to write schemas in Tcl. The commands translate the class definitions read from a schema definition file into procedure calls to the meta-schema facility. They also automate the demand-loading of compiled modules into the execution protocol engine. Here is the grammar for the schema definition language, using the syntax notation introduced in Section 3.2:

```
rule tcl_command {
   tcl_builtin_command
  | "enc_enum" IDENTIFIER "{" repeat { STRING_LITERAL } "}"
  | "enc_class" IDENTIFIER opt { module:IDENTIFIER }
    "{"  opt { "inherit" repeat { class:IDENTIFIER } }
         list { class_member }
    "}"
}
rule class_member {
   "constructor" c_proc
  | "destructor" c_proc
  | { "method" | "proc" } type_designator IDENTIFIER
    "{" list { variable } "}" c_proc
  | "public" type_designator IDENTIFIER get:c_proc opt { set:c_proc }
}
rule type_designator {
   "void" | "int" | "float" | "boolean" | "string" | type:IDENTIFIER
}
```

What is most remarkable about this schema language is the handling of demand-loadable compiled modules as well as the use of "C" function names to associate procedures and data members with the appropriate wrapper functions. To automati-

cally load a compiled module, a class definition may optionally name the location of this module in the UNIX file system. A table of already loaded modules is maintained so that a module is loaded only once. When the module name is omitted, the corresponding class definitions are assumed to be compiled into the engine. Naming "C" functions works in both cases. Whereas only fixed numbers of positional parameters are supported for ordinary methods, constructors and destructors are allowed to have arbitrary, named parameters. The schema programming interface allows both alternatives for either type of function.

### 4.4.2 Object use

Once the engine has a description of the classes defined in a compiled module, objects can be created from classes that define a constructor. In the interpreter into which the schema has been read, each class defines a new Tcl command named after the class. New commands are also created for static member functions, named with class name and member name separated by "::". The class commands are used to create new instances of the associated compiled objects with the following basic command syntax:

```
rule class_command {
   class:IDENTIFIER { object:IDENTIFER | "#auto" }
   list { named_parameter }
  | class:IDENTIFIER "info" { "inherit" | "heritage" }
}
rule named_parameter {
   "-" parameter:IDENTIFER[*] typed_value
}
rule typed_value {
   INTEGER_LITERAL | FLOAT_LITERAL
  | BOOLEAN_LITERAL | STRING_LITERAL
  | enum:STRING_LITERAL | OBJECT_HANDLE }
}
```

The result of executing a class command is a new object handle, either automatically generated or chosen by the programmer. In addition a new command is created in the Tcl interpreter that is used to invoke methods on the new object. A class

[*]: no space is allowed between the dash "-" and the parameter:IDENTIFIER

command may fail if the underlying constructor function signals an error, e.g. because of conflicting lock requests. It is the responsibility of the wrapper function for the constructor to check the validity of any named parameters passed on the command line. A class command may also be used to retrieve the direct ancestors of the associated class from the schema data structure.

The purpose of invoking a constructor through a class command is not always to create an entirely new object. As the execution protocol engine maintains no persistent memory in itself, a constructor call is always needed to pull an object from a compiled module into the realm of execution protocols. In these cases the named parameters allowed on the constructor command line may pass selection criteria like names to the constructor implementation. The check-out of a design object in Nelsis could be regarded as a typical example of this. The Nelsis DMI returns a CELL_HANDLE which is used as an object identifier in the wrapper module. The destruction of the wrapped object using the standard message *destroy* on an object handle may then actually check in the design object. This application shows why destructors may have parameters in execution protocols, too. Nelsis requires a completion code to accompany the check-in operation.

Object handles need not be created with class commands. They can either be returned from procedures or be referenced through a data member of a wrapped object. The object handles are created the first time the procedure is invoked or the data member is accessed. The object handle can be used as value wherever an object of that type is expected. In any case, its value is also the name of a new Tcl command that is used to invoke methods on the associated object:

```
rule object_command {
   object:IDENTIFIER "destroy" list { named_parameter }
  | object:IDENTIFIER opt { ancestor:IDENTIFIER "::"[*] }
    method:IDENTIFIER list { positional_parameter }
  | object:IDENTIFIER "config"
    repeat { "-" attribute:IDENTIFIER[*] typed_value }
  | object:IDENTIFIER "info"
    { "class" | "inherit" | "heritage" | "proc" | "method" | "public" }
  | object:IDENTIFIER "isa" class:IDENTIFIER
}
rule positional_parameter { typed_value }
```

[*]: no space is allowed between the ancestor class name, the two colons "::", and the method name

Methods and data members are searched in the class of the object first and then, if not found, recursively upwards the inheritance hierarchy of the object class. A list of class names in exactly the sequence of method lookup can be retrieved by issuing the request *info heritage*. This default search can be overridden by prefixing a method name with the base class that defines the requested method. Data members can be set with the standard message *config*. A method with the name of the data member is automatically provided to read the value of a data member. Other standard messages exist to query the type data structures for certain information. The standard message *destroy* invokes the destructor with optional parameters. A short example illustrates the use of class commands and the manipulation of objects:

```
/* file: class.h */
struct base;
struct derived;
struct base {
  int a;
  char* s;
  base* obj;
  base(): a(0), s(0), obj(0)      { printf ("base (0)\n"); }
  base (char* s_): a(0), s(strdup(s_)), obj(0)
                                  { printf ("base (s='%s')\n", s); }
  int incr (int i)                { printf ("incr(%d)->%d\n", i, a+i);
                                    return a += i; }
};
struct derived: public base {
  int b;
  char* t;
  derived(): b(0), t(0)           { printf ("derived (0)\n"); }
  derived (char* s_): b(0), t(0), derived (s_)
                                  { printf ("derived (s='%s')\n", s); }
};
int add (base* a, base* b)      { return a->a + b->a; }
```

```
/* file class.tcl */
enc_enum blubber { blubber waber glibber }
enc_class base libclass.so.1.0 {
 constructor                              new_base
 destructor                               del_base
 method int incr { {int incr} }           base_incr
 public int a        base_get_a           base_set_a
 public blubber soft base_get_a           base_set_a
 public string s     base_get_s           base_set_s
 public base obj     base_get_obj         base_set_obj
}
enc_class derived libclass.so.1.0 {
 inherit base
 constructor                              new_bar
 destructor                               del_bar
 public int b        derived_get_b        derived_set_b
 public string t     derived_get_t        derived_set_t
}
enc_class ex libclass.so.1.0 {
 proc int add { {base a} {base b} }       ex_add
}
```

Now we can create some objects:

```
% base b -s “aBase”        -> base (s=”aBase”)
% derived d -s “aDerived”  -> base (s=”aDerived”)
                              derived (s=”aDerived”)
% b config -soft glibber; d config -a 40
% ex::add b d              -> 42
```

The last line shows the use of a static member function. Please note that enumeration values are quietly coerced into integers.[*]

[*] The result of this addition operation is of course purely coincidental and has no philosophical meaning whatsoever [Adams 79].

### 4.4.3 An example: The Nelsis DMI

The following is an excerpt from a schema definition for the Nelsis DDM system:

```
enc_enum completionModeEnum {quit complete}
enc_enum openModeEnum {read write update}
enc_enum purposeEnum {edit import derive export extract view}
enc_class DesignObject {}
enc_class Library $DDMLib/schemas/libnelsis.so.4.3 {
 constructor                                  new_Library
 destructor                                   del_Library
 method string query {{string query}}         Library_query
 method DesignObject designObject {
     {purposeEnum purpose} {string name} {string viewType}} \
                                              Library_designObject
 ...
}
enc_class DesignObject $DDMLib/schemas/libnelsis.so.4.3 {
 destructor                                   del_DesignObject
 public Library lib                           DesignObject_getLib
 public string mod_name                       DesignObject_getModName
 public int v_number                          DesignObject_getVNumber
 public string view_type                      DesignObject_getViewType
 public string hierarchy                      DesignObject_getHierarchy \
                                              DesignObject_setHierarchy
 ...
}
enc_class Nelsis $DDMLib/schemas/libnelsis.so.4.3 {
  proc void init {{string tool}}              Nelsis_init
  proc void quit {}                           Nelsis_quit
  proc void checkInAll {}                     Nelsis_checkInAll
 ...
}
```

Only some of the class members are shown here to demonstrate the use of the schema definition language. The data member *hierarchy* of class DesignObject is worth noting. Nelsis allows to access hierarchical relationships through the use of streams. The two wrapper functions *DesignObject_getHierarchy* and

*DesignObject_setHierarchy* map this stream interface to an interface to a pseudo data member *hierarchy*. Whenever there is read access to this data member, the corresponding stream is read and its contents is returned as value of this data member. On write access, the string passed as parameter to the *set* procedure is written as new contents into the *hierarchy* stream. The class *Nelsis* collects functions that are not associated with any particular class. As such its member functions are static and do not pass an object identifier as parameter. A typical use of these classes in an execution protocol would look like this:

```
Nelsis::init browser_tools
Library lib -path ~/projects/DP32 -mode update
browse “DP32”
lib destroy -mode complete
Nelsis::quit

proc browse {mod_name {lv 0}} {
 puts “[incr lv]: mod_name”
 set q [lib query “GET DesignObject
        WHERE      Module ITS Name == $mod_name AND
                   Module ITS ViewType == ’structure’”]
 if { [llength $q] == 1 } return
 set do [lindex $q 1]
 foreach h [$do hierarchy] { browse $h $lv }
}
```

This short Tcl program will open the Nelsis project “~/project/DP32” for update, and list the elements of the hierarchical composition of a *Module* “DP32”. All the real work is done by the procedure browse, which prints the *Module* name passed as parameter and then searches for a *DesignObject* with *Module* name “DP32” and *ViewType* “structure” in the Nelsis database. The method *query* of class *Library* provides the interface to the Nelsis DML. It allows to pass queries as strings and returns query results as a Tcl list that are easily processed by Tcl commands. The first list element always contains the list of attributes requested, followed by a - possibly empty - list of results. The first *DesignObject* found is extracted from the list. A loop then iterates over its children in the composition hierarchy returned as value from the pseudo data member *hierarchy*, calling itself recursively with the *Module* name of that child.

# 5. Design information management

## 5.1 Overview

In this section we describe the design information management (DIM) service of an EDA framework. A conceptual schema is presented that fulfils the requirements on design structure management summarized in Section 1.6*. It can be used by tightly integrated and encapsulated design tools to access design objects and by designers to query the state of their design. The conceptual schema is associated with an application programming interface used by design tools and framework services such as the encapsulation service described in Chapter 4. The application programming interface in turn is implemented by invoking functions from an underlying data manager to access design objects, their structure and representation. In Section 5.4 we explain different approaches for an implementation of a DIM service and illustrate these approaches with two case studies conducted on the ObjectStore OODBMS and on the Nelsis CAD Framework. While the implementation on an OODBMS is easier, implementing the new service on an existing CAD framework has two advantages, assuming the framework already is the basis for a complete design environment with design tools:

1. Design tools encapsulated using the new service can inter-operate with tools already integrated by other means.
2. Designers can stick with a proven and familiar design system and need not be trained to use a completely new system.

## 5.2 Conceptual schema

The conceptual schema described in this section allows to configure a DIM service to manage structural information about design objects (**R1**). We use the Xplain semantic data modelling technique [terBekke 92] introduced in Section 3.1. The schema will not be presented as a whole but rather developed incrementally. The diagrams will sometimes not show all attributes of the depicted types. The textual type definitions, however, are complete. In each step, we focus on a specific aspect of structural design information, identify relevant concepts and formalize them using the Xplain notions of *types*, *aggregation,* and *specialization*. As new types are

*. Bracketed marks **R1**..**R7** refer to this list of requirements

defined in terms of other types, previously defined types may be redefined to avoid redundancies.

### 5.2.1 Interfaces, implementations, and configurations

As an electronic design project proceeds, the specification of a designed module is refined stepwise; its interface evolves according to the levels of detail, each having a number of implementations in different domains. In our schema, a module represents an electronic system and aggregates a set of design objects that all describe the same electronic design. It forms the root of an inheritance tree of refined interfaces on various levels of detail which in turn have implementations in different domains. Interfaces are arranged in an inheritance tree where interfaces at low levels of detail are situated near the root and refined interfaces at high levels of detail can be found near the leaves. No dedicated relationship is reserved for the representation of this inheritance hierarchy. Rather, we use equivalence relationships between the interfaces in which the more abstract design objects assume the role of sources and the more refined design objects assume the role of targets (Figure 12).

**type** Date: INTEGER.
**type** Designer: STRING.
**type** Name: STRING.
**type** DesignObject = Date, Designer, Name.

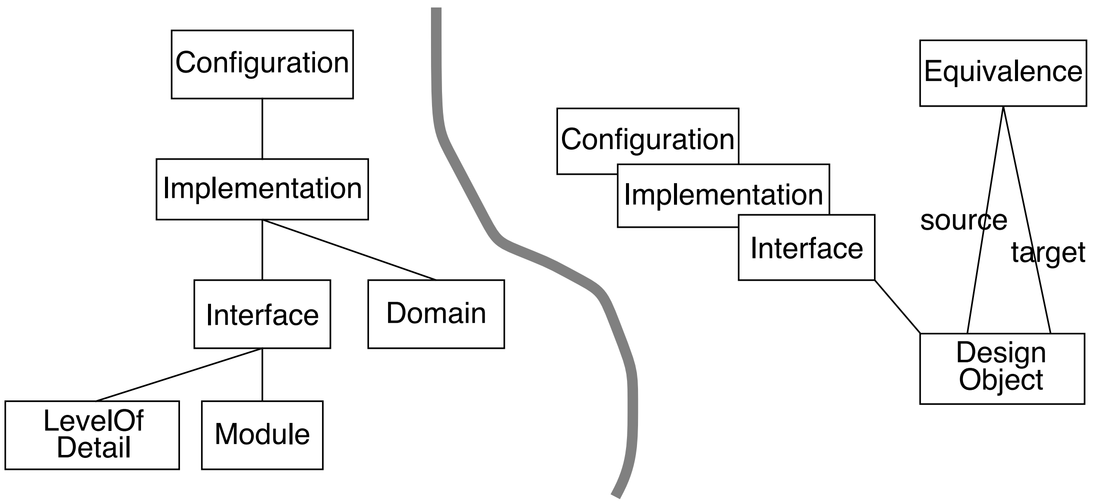


***Figure 12.*** *Interfaces, implementations, and configurations.*

According to our definition in the Introduction, a design object describes an electronic design. However, the type *DesignObject* defined here does not have an attribute to store a design description. The capability to be associated with design representation information is not represented explicitly in our schema. It is instead hard-coded into the framework module that provides design information management services to design tools through its programming interface.

```
type Class: STRING.
type Tool: STRING.
type Equivalence = source_DesignObject, target_DesignObject,
     Class, Tool.
```

Equivalence relationships are very versatile. Here, they are used to arrange interfaces in an inheritance tree.[*] It is not trivial and generally depends on the nature of the design domain to check if two design objects are equivalent with respect to a certain property (or, phrased differently: if the design objects belong to the same *Equivalence Class*). This somewhat vague notion of equivalence makes us introduce explicit equivalence relationships and not equivalence sets as proposed in [Katz 86]. Equivalence relationships are thought to be established by tools and not automatically by the DIM service. Equivalence transitivity is therefore introduced on a case-by-case basis by establishing explicit equivalence relationships between design objects and is not answered in advance by some global property of equivalence sets. Our use of equivalence relationships follows their use in the Nelsis schema as motivated in [vanderWolf 93], p. 88.

```
type LevelOfDetail:
  { 'architecture', 'algorithm', 'functional_block', 'logic', 'circuit' }.
type Domain: { 'functional', 'structural', 'physical' }.
type Module = Name.

type Interface = [DesignObject], LevelOfDetail, Module.
type Implementation = [DesignObject], Domain, Interface.
type Configuration = [DesignObject], Implementation.
```

[*]. A potential danger of this flexibility is that possibly object properties are not properly represented in the schema but rather are mapped to obscure equivalence relationships, making it hard for the designer to understand the disposition of his design.

Interfaces, implementations, and configurations are special design objects. As such, they inherit the capability to be associated with design representation information. They are arranged in a “static” (i.e., fixed by the schema) access hierarchy. Relatability ensures that whenever we want to create a configuration for a module, a corresponding implementation and interface have been defined first.

With this schema in mind, we regard a module as a set of design objects all describing the same electronic system from different view-points. The set of design objects is partitioned into three disjoint subsets according to the role the elements of each set play in the design description of an electronic system. Objects in the interface subset describe the appearance of an electronic system viewed from the outside. Each interface is related to a set of implementations that describe different approaches as to how this outside view is accomplished technically. An implementation itself may use components specified by another module. Which design description of either component exactly is used is described by design objects in the configuration subset. Again, there may be different alternative configurations for each implementation. This schema satisfies requirements **R2**, **R3**, and **R5**.

### 5.2.2 Projects

The initial schema introduced in the previous section does not yet meet the requirement that all design data must not be contained in a global pool. We introduce the concept of *projects* to structure the global object pool:

**Definition:** A *project* is a local environment in which design activities may be performed. [vanderWolf 93], p. 71

Projects allow to structure the global object pool so that design activities can be performed locally without effecting other projects. A production design environment would have to go even further and allow projects to be hierarchically composed of other projects. Each level of hierarchy introduces an additional level of privacy in which projects near the root of the hierarchy represent published designs and completed design libraries. In such a scenario, a designer uses a local project as a workspace in which he performs experimental design steps. He can use locally created objects as well as ones imported from further up in the project hierarchy. Once a set of design objects is stable, it is checked into a parent workspace and thus made public for other designers.

We will not introduce a formal schema for projects as this concept is not directly related to the problem of design tool encapsulation. An implementation nevertheless will have to add the concept of projects. Both our implementation bases offer projects either directly (Nelsis, [Dimes 93a], p. 13) or in the form of a hierarchy of workspaces (ObjectStore, [ObjectDesign 94], p. 15).

### 5.2.3 Composition hierarchy

A useful means to reduce the complexity of an electronic design is to compose a module of smaller components, that in turn may again be composed of sub-components. Hierarchy relationships are established between a compound implementation and the components it uses. As each component may be instantiated more than once in an implementation, we need to keep track of these instances, too. Every instance is associated with a constructor that specifies the exact circumstances under which it is used in its parent.

Components do not reference the used design object directly but via component configurations. Component configurations bind a component to a child design object (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” (Figure 13).

```
type DesignObject = Date, Designer, Name.
type Interface = [DesignObject], Module, LevelOfDetail.
type Implementation = [DesignObject], Interface, Domain.
```

These are the same definitions for *DesignObject* and its subtypes as in the previous section.

```
type Constructor: STRING.
type Component = parent_Implementation, Name.
type Instance = Constructor, Component, Name.
```

Components establish hierarchy relationships between a parent implementation and another design object. We explicitly distinguish between components and instances to accommodate different uses of the information that is manageable by this schema. Design tools generally need the exact instantiation information with details

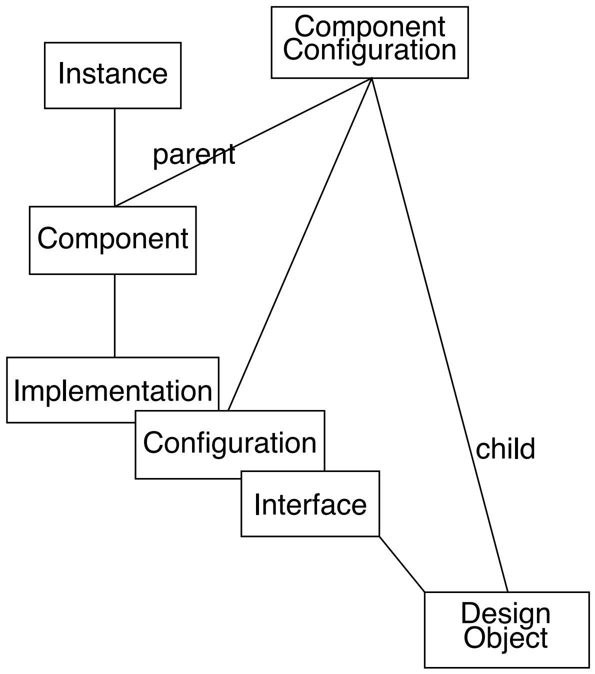


***Figure 13.*** *Composition hierarchy. The relationships between the subtypes of* DesignObject *are not shown.*

about each instance. On the other hand, the encapsulation service only needs to know which design objects are used in which implementation. This information is represented by components.

```
type Info: STRING.
type Configuration = [DesignObject], Implementation.
type ComponentConfiguration =
     Component, child_DesignObject, Info, Configuration.
```

Component configurations establish the actual *binding* between a component and the design object that realizes it. This added level of indirection allows to manage a set of configurations as described above. With these definitions, we satisfy requirements **R4** and **R5**.

Already more complex than a model for composition hierarchies without configurations, this model still does not allow to individually configure each occurrence of a sub-module in a composition hierarchy. If we wanted to model the configuration of individual occurrences as well, we would have to modify the type definitions as follows:

```
type Configurable = Name.
type ComponentConfiguration = Configurable, child_DesignObject,
     Info, Configuration.
type Component = [Configurable], parent_Implementation.
type Instance = [Configurable], Constructor, Component.
```

With this model, it is possible to individually configure each instance in an implementation. Although this amount of control may be desirable for purposes such as back-annotation, the introduction of additional components with different characteristics can still be applied when using the simpler model. In the sequel, we therefore choose the simpler model as depicted in Figure 13.

Figure 33 on page 176 depicts the composition hierarchy of the design description for the DP32 test bench circuit, including two configurations. The large diamonds in the figure represent configuration objects, the small diamonds represent component configurations. The links between configurations and component configurations are not shown. Rather, the two configurations and their respective component configurations are rendered in two different shades. As can be seen in this hierarchy graph, components never reference their child design objects directly but do so via component configurations.

A single configuration only references component configurations that configure components of a single parent implementation. Formally, the following assertion holds for all component configurations:

```
assert ComponentConfiguration its
  Configuration its Implementation == Component its Implementation
```

A single configuration may nevertheless configure components more than one level down the composition hierarchy. This is achieved by having component configurations specify a configuration, not an implementation or interface as child design object.

Consider, for example, the following excerpt from the configuration declaration for the DP32 test bench:

```
configuration dp32_rtl_test of dp32_test is
 for structure
     for cg: clock_gen
         use entity work.clock_gen (behaviour)
         generic map (Tpw => 8ns; Tps => 2ns);
     end for;
     for mem: memory
         use entity work.memory (behaviour);
     end for;
     for proc: dp32
         use entity work.dp32 (rtl);
         for rtl
             for all: reg_file_32_rrw
                 use entity work.reg_file_32_rrw (behaviour);
             end for;
             for all: mux2
                 use entity work.mux2 (behaviour);
             end for;
             -- more component configurations
         end for;
     end for;
  end for;
end dp32_rtl_test;
```

First, we note that in the configuration of dp32_test.structure the instances are configured individually. With our simplified model of component configurations we cannot in principle represent this fine “configuration granularity”. As, however, each component is only instantiated once, we can safely replace the instance labels (*cg*, *mem*, *proc*) with *all* qualifiers without losing information. In more complex cases, the introduction of additional components would have been necessary.

In this example, a configuration for the register-transfer level implementation of the DP32 microprocessor is embedded in the configuration for the test bench circuit. Within this nested configuration, components are configured, this time directly using *all* qualifiers. The nesting of configurations resembles the logical nesting of submodules. Without losing information, we can unwind (“flatten”) the configuration nesting by introducing explicit configuration declarations which are then referenced as children by the component configurations:

```
configuration dp32_rtl_test of dp32_test is
 for structure
        for all: clock_gen
                use entity work.clock_gen (behaviour)
                        generic map (Tpw => 8ns; Tps => 2ns);
        end for;
        for all: memory
                use entity work.memory (behaviour);
        end for;
        for all: dp32
                use configuration work.dp32_rtl;
        end for;
  end for;
end dp32_rtl_test;
configuration dp32_rtl of dp32 is
 for rtl
        for all: reg_file_32_rrw
                use entity work.reg_file_32_rrw (behaviour);
        end for;
        for all: mux2
                use entity work.mux2 (behaviour);
        end for;
        -- more component configurations
 end for;
end dp32_rtl;
```

We can directly map this flattened configuration into an object graph according to our conceptual schema (Figure 14).

Although configuration *dp32_rtl_test* was our starting point, the configuration *dp32_rtl* originally nested within *dp32_rtl_test* now is a first class object of type Configuration and can be referenced from component configurations other than those contained in *dp32_rtl_test*. While simplifying the model, flattening nested configurations therefore simplifies design reuse.

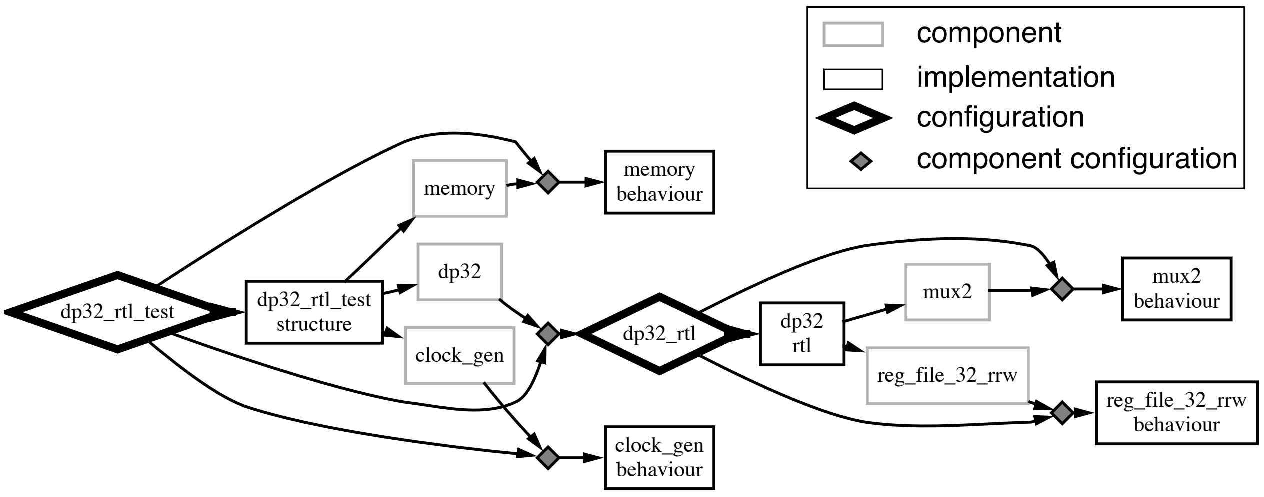


***Figure 14.*** *Object graph for the dp32_rtl_test configuration. Interface and info objects are left out for simplicity.*

### 5.2.4 Version derivation and design transactions

The relationships introduced in this schema are crucial to design management. However, we do not have to develop something new here as e.g. the schema of the Nelsis CAD Framework fits our needs well. We therefore take the following type definitions directly from the Nelsis schema [vanderWolf 93], only slightly extending the definition of *DesignObject*.

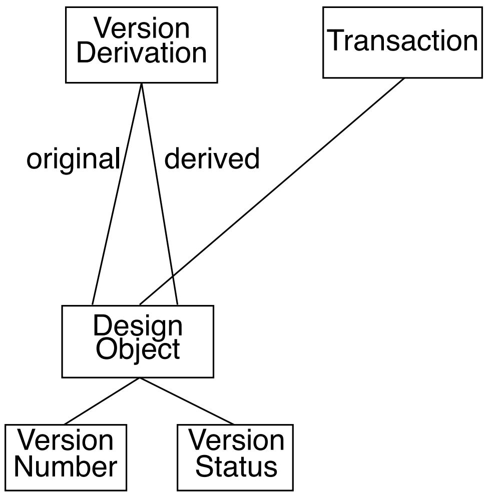


***Figure 15.*** *Version derivation, design transactions, and equivalence.*

```
type  Name: STRING.
type  Date: INTEGER.
type  Designer: STRING.
type  Language: STRING.
type  VersionNumber: INTEGER.
type  VersionState: { ‘backup’, ‘derived’, ‘working’, ‘actual’ }.
type  DesignObject =
      Date, Designer, Name, VersionNumber, VersionState.

type  VersionDerivation = original_DesignObject, derived_DesignObject.
type  CompletionMode = { ‘running’, completed’, ‘aborted’ }.
type  Tool: STRING.
type  Transaction = Tool, CompletionMode.

type  Module = Name, LastVersion.
```

The definition of attributes for version number and version state emphasizes the central role *DesignObject* plays in our schema. Design objects are the only entities that can be versioned. This property, of course, is inherited by its subtypes *Interface*, *Implementation*, and *Configuration*.

Version derivation adds a further structuring mechanism to the set of design objects by allowing to relate a design object with a set of derived design objects and a derived design object with one or more originals. Note that in the case of more than one original a merge must have occurred to weed out potential incompatibilities introduced on the two branches. Design objects are versioned relative to a module. To uniquely identify objects within the module (other than by their object identifier), they are automatically assigned a version number on creation. In addition, a design object is assigned a version status for version selection in cases in which configurations are not used.

Design objects are involved in potentially long design transactions. Information about completed transactions and associated design objects is not simply deleted on transaction commit but is stored to render a complete design history.

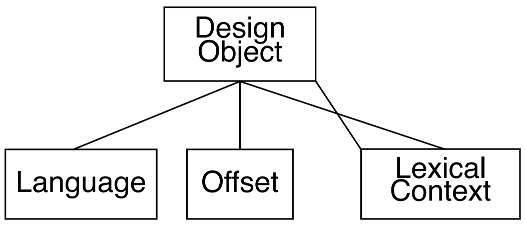


*Figure 16. Design objects and lexical contexts.*

### 5.2.5 Contexts

Encapsulated design tools access design information through design files. The design information manager, however, manages design objects. A mapping between design files and design objects is therefore necessary. To support this mapping, the structural information is enhanced with "contexts", objects that reflect the nesting of design objects in design files.

Each design object is associated with a textual design representation in a selected design description language. In addition, it is also associated with the lexical context in the design file from which it originates and the offset in this context. In more complex situations, the design object itself may again play the role of a lexical context for some sub-ordinate design object (Figure 16). Figure 17 gives a schematic example; in Figure 18, lexical contexts are applied to a piece of VHDL description.

In more complex languages or if the design methodology dictates a more fine-grained management of design objects, a refinement of lexical contexts is called for. Lexical contexts only allow to manage the lexical structure of a hierarchy of contexts, basically by storing offsets into the context of a given design object. If the con-

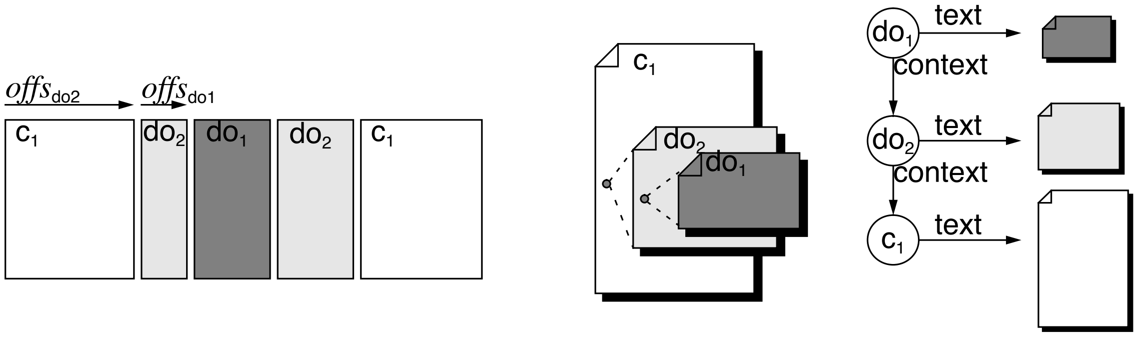
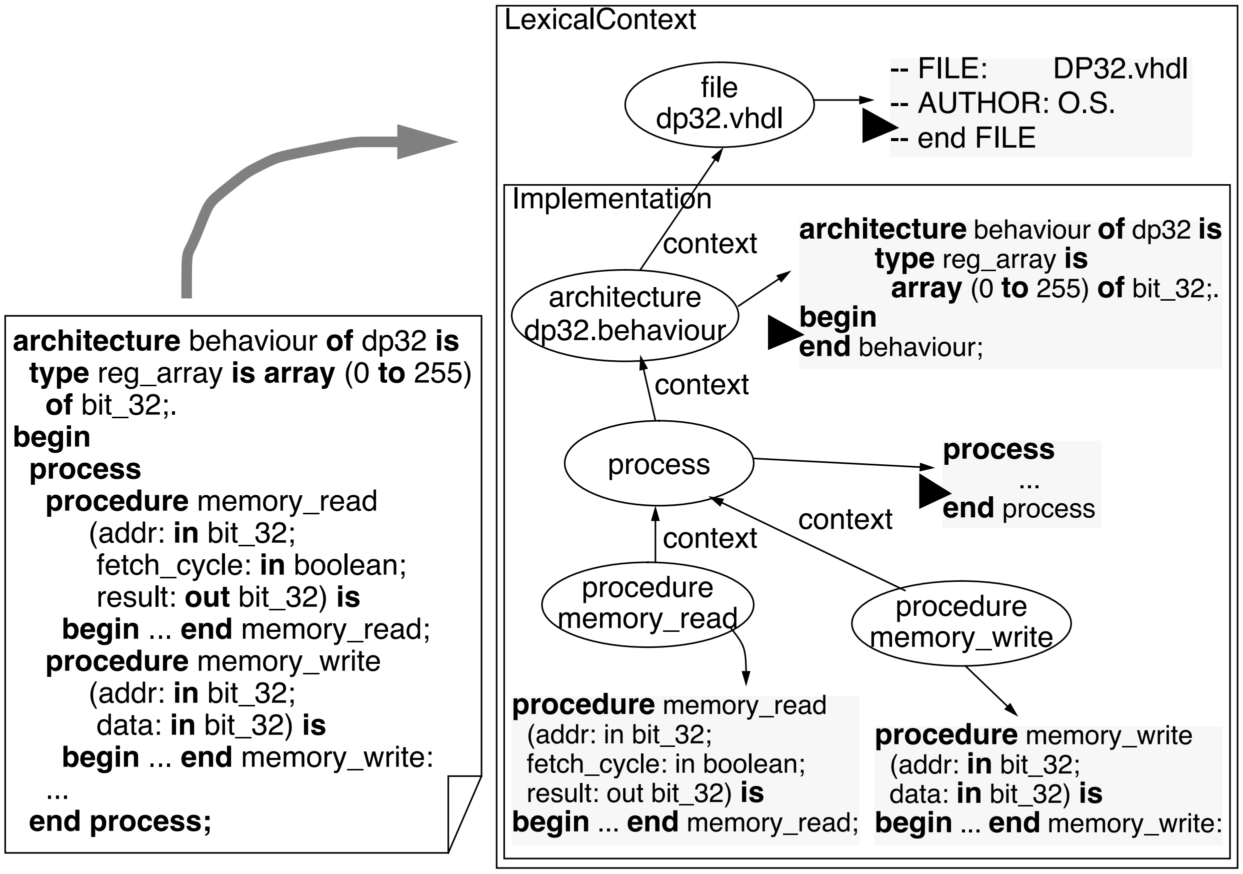


*Figure 17. Nested contexts. The sequentialized, textual representation of nested contexts is mapped to a tree in the DIM manager where a single context may embrace any number of design objects, nested to arbitrary depth.*



```
architecture behaviour of dp32 is
  type reg_array is array (0 to 255)
    of bit_32;.
begin
  process
    procedure memory_read
          (addr: in bit_32;
           fetch_cycle: in boolean;
           result: out bit_32) is
      begin ... end memory_read;
    procedure memory_write
          (addr: in bit_32;
           data: in bit_32) is
      begin ... end memory_write:
    ...
  end process;
```

```
architecture behaviour of dp32 is
        type reg_array is
           array (0 to 255) of bit_32;.
begin
end behaviour;
```

```
procedure memory_read
  (addr: in bit_32;
  fetch_cycle: in boolean;
  result: out bit_32) is
begin ... end memory_read;
```

```
procedure memory_write
  (addr: in bit_32;
  data: in bit_32) is
begin ... end memory_write:
```

***Figure 18.*** *Using lexical contexts to represent parts of the behavioural architecture specification of the DP32 processor. The shaded areas are textual design representation, with the insertion points of children marked with a* ▶ *. The two boxes denote the extents of the types* LexicalContext *and* Implementation. *Note that* Implementation *is-a* DesignObject *is-a* LexicalContext, *so every* Implementation *object is also a* LexicalContext.

text itself is a *DesignObject*, it may be modified as such, invalidating the offsets of design objects lexically nested within it. An obvious solution to this problem is not to store a numeric offset with the child but rather to insert a special marker into the parent. This marker may be moved around freely, and, as long as it is not deleted, may be replaced by the text of the child on export. A disadvantage of this approach is that the parent context must be searched for markers to insert the text of children.

A further refinement is to actually retain the syntactical structure of the parent with respect to its children. An extension of the schema shown in Figure 16 accomplishes this (Figure 19):

```
type LexicalContext = Language, Date, Designer, Name.
type DesignObject = [LexicalContext], LexicalContext, Offset.

type ParseTreeConstructor: STRING.
type SyntacticalDesignObject = [DesignObject], ParseTreeConstructor.
```

This schema answers two questions. (1) How are design objects embedded in their lexical context, and (2) how does the design object lexically or syntactically contain other design objects? Every design object is embedded in a lexical context. This results from the fact that we extract design objects from design files. There, the file (stripped of the actual representation of the design object) is the lexical context. There are two ways in which the design representation of textually nested design objects can be managed by the design information manager. In the simpler approach, a design object itself plays the role of lexical context for a set of embedded design objects as described above. In more complex scenarios, not the lexical but the syntactical structure of a design object is managed by the design information manager. A specialized form of *DesignObject* carries a parse tree constructor. A parse tree constructor is a piece of Tcl code that describes how to construct a parse tree from the values of a design object's attributes and those of related objects. In Section 6.5 on page 114 we will explain in detail what a parse tree constructor looks like and how it can be constructed largely automatically from a language specification.

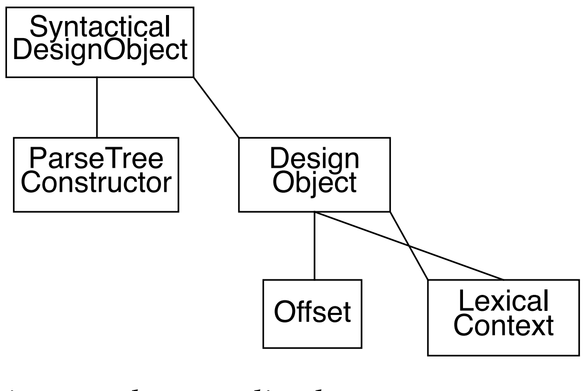


***Figure 19.*** *Design objects and generalized contexts.*

## 5.3 The design information programming interface

### 5.3.1 Overview

The programming interface for our design information management component has to support the following tasks:

1. browse the design structure
2. find a design object based on its name
3. access the design representation associated with a design object
4. access design hierarchy by using configurations

While tasks in (1) and (2) only imply a short read or write access to design structure, tasks in (3) and (4) usually involve an interactive editing session or a lengthy design tool run. According to this distinction, different kinds of operations have to be employed for these tasks:

- (1) and (2) only require short operations on design structure information,
- tasks in (3) require long operations on both design structure and design representation.

In addition, tasks in (4) allow to access a single design hierarchy "filtered" by a configuration; components encountered in a design hierarchy are bound to the design objects determined by the configuration. For hierarchy access special operations are provided that perform this filtering.

### 5.3.2 Accessing design structure

The operations to access design structure can be directly derived from our conceptual schema. A principal design decision is whether to provide a procedural interface or a query language. The Xplain semantic data modelling technique defines a query language that could be used directly as an access method to a design information management service in a framework. A query language is a good choice as an interface paradigm when interactive access is desired. It is, however, more difficult to implement, especially when portability across various design data managers is an issue. It also does not blend naturally into the code of a design tool written in "C" of "C++". We therefore choose to define a procedural interface to our design information management service in the programming language “C++”. A query interface can still be constructed on top of it.

Objects of aggregate types are accessed through object references. Such a reference may be the actual memory address of an object or a database identifier. Overloading in “C++” allows to handle both cases in a uniform and transparent way. Object references are only valid in the scope of a *project*. Before any other operation is invoked, a project handle has to be obtained by invoking the static member function

```
static dim_Project* dim_Project::open
  (string name, dim_OpenMode openMode);
```

*OpenMode* can either be *read* or *update*. For a given project, only a single project handle in *update* mode may be requested at any one time. Of course, multiple *read* handles may be requested. A project handle is released by a call to

```
void dim_Project::close (dim_CloseMode);
```

*CloseMode* may either be *commit* to request that all changes are to be made permanent or *abort* to discard all changes made with this project handle. The interface is schematically derived from our conceptual schema. Base types are mapped to the corresponding base types in “C++”:

| Xplain base type | mapped to C++ type |
|---|---|
| INTEGER | int |
| REAL | double |
| STRING | char* |

***Table 9.*** *Mapping of Xplain base types to C++ types*

For every Xplain definition

```
type BaseType: base-type.
```

there is a corresponding type definition in “C++”:

```
typedef base-type BaseType;
```

The definition of an aggregate

```
type Aggregate = role_Component.
type Component = ... .
```

is mapped to two class definitions in “C++”:

```
struct Aggregate {
  Component* role_component();
  void role_component (Component*);
};
struct Component {
  Set<Aggregate*> aggregates();
  ...
};
```

Please note that for each component there are two access functions, one to read a value and one to write a value. We do not map components directly to data members to have more control over the underlying implementation. For all practical purposes, a pair of access functions is equivalent to a data member. Access functions, however, are free to retrieve the actual data values from an underlying data manager or cache data values for greater efficiency.

Access functions automatically maintain consistency. In the above example, whenever a component is set for an aggregate, the aggregate is automatically inserted into the aggregate set in the component.

Access from a component to its aggregate is accomplished by *set* functions. The usual bunch of functions is provided to iterate through the elements in the set and to insert and delete members from the set. Only one function is needed to access the actual set.

A specialization in Xplain is mapped to class inheritance in “C++”. For example, the following definition of a tree structure in Xplain

```
type Node = Value.
type InnerNode = [Node], parent_Node.
```

is mapped to the following “C++” classes:

```
struct InnerNode: public Node {
  Node* parent_node();
  void parent_node (Node*);
};
struct Node {
  Set<InnerNode*> inv_parent_nodes();
};
```

With these mappings it is possible to access all the objects managed by the design information manager. The only thing missing is a way to retrieve an initial object reference. This can be accomplished by accessing an object by its name with the following member functions of *dim_Project*:

```
Set<Module*> getModule (string moduleName);

Set<Interface*> getInterface (
  string moduleName, string interfaceName, string levelOfDetail);

Set<Implementation*> getImplementation (
  string moduleName, string interfaceName,
  string implementationName, string domain);

Set<Configuration*> getConfiguration (
  string moduleName, string interfaceName,
  string implementationName, string configurationName);
```

These functions return sets of design object references, because object names need not be unique within a project. The sets can be traversed to find the desired object.

### 5.3.3 Accessing design representation

As mentioned above, access to design representation is of a different nature. The operations to access design representation need to be associated with a transaction that provides protection against concurrent write accesses and preserves consistency by maintaining duplicates for read accesses while a write transaction is in progress. The following functions are provided to manipulate design objects in the context of a design transaction. All functions work on *design object handles*, objects of class *dim_DesignObject* and its sub-classes *dim_Interface*, *dim_Implementation*, and *dim_Configuration*. The following member functions of *dim_Project* are used to create design objects. The attributes *Designer, Date, VersionNumber, VersionState, Language, LexicalContext*, and *Offset* of Design Object are filled in automatically either with context information or with default values by the function implementation:

```
dim_Interface* createInterface (
  string moduleName, string interfaceName, string language,
  string levelOfDetail);

dim_Implementation* createImplementation (
  string moduleName, string interfaceName, string implementationName,
  string language, string domain);

dim_Configuration* createConfiguration (
  string moduleName, string interfaceName,
  string implementationName, string configName);
```

If the named object does not yet exist, a new design object is created. If an object of this name already exists, a new version of it is created with version status *working*. In addition, lexical contexts are created with the function

```
dim_LexicalContext* createContext (
  string contextName, string language);
```

Normal contexts are not manipulated in an interactive design transaction. Rather, they are dynamically constructed from parse tree templates at the time of export (cf. Chapter 6 on page 103).

The following member functions of *dim_Project* open existing design objects:

```
dim_Interface* dim_Project::openInterface (Interface*, dim_OpenMode);
dim_Implementation* dim_Project::openImplementation (
  Implementation*, dim_OpenMode);
dim_Configuration* dim_Project::openConfiguration (
  Configuration*, dim_OpenMode);
```

These functions yield a handle for the corresponding object type, opened for *read* or *update*. After manipulations on any design object, the changes made have to be committed with the function

```
void dim_DesignObject::close (dim_CloseMode closeMode);
```

This call creates a new version that is derived from the original. The design object reference for a design object handle can be accessed by the function

```
DesignObject* dim_DesignObject::id();
```

The actual design data, in turn, are accessed through a stream-based interface:

```
dim_Stream* dim_DesignObject::openStream (
  string name, dim_OpenMode);
dim_Stream::read (int count);
dim_Stream::write (string buffer, int count);
void dimStream::close (dim_CloseMode);
```

Once a stream is opened, operations can be applied to get its size and to read and write its contents. To conclude, we have the following hierarchy of calls that reflects the transaction hierarchy needed to access the actual design data:

```
dim_Project::open();
    dim_Project::openDesignObject();
        dim_DesignObject::openStream();
            // stream access
        dim_Stream::close();
    dim_DesignObject::close();
dim_Project::close();
```

### 5.3.4 Hierarchy and configurations

A configuration collects all the component configurations for its implementation. The function

```
Set<dim_InstanceInfo*> dim_Configuration::getInstances
  (dim_BindingMode bindingMode);
```

returns a set of instance information structures. These are an abstraction of the instance type directly derived from the conceptual schema, conveniently hiding the component configurations from the programmer:

```
struct dim_InstanceInfo {
  string name;
  string constructor;
  DesignObject* child;
};
```

With binding mode *static*, for each instance, this structure references the child design object currently bound by the configuration. With binding mode *dynamic*, for each instance, an implementation object is selected:

- If the child design object points to a configuration, this configuration is selected.
- If the child design object points to an implementation, this implementation is selected.
- If the child design object points to an interface and an implementation with version status *actual* exists for this interface, it is selected. Otherwise, an implementation with version status *working* is selected. If this also fails, the implementation with the highest version number is selected.

To define component configurations in an opened configuration, the function

```
void dim_Configuration::setInstances (Set<dim_InstanceInfo*>)
```

is used. Here, the set of instance information structures has to be constructed first and is passed to the function as a parameter.

Whereas the operations presented in Section 5.3.2 represent a low-level interface, the operations of Sections 5.3.3 and 5.3.4 work on lower levels of detail. Whenever a suitable, more abstract function is available to accomplish a certain task in design information management, it should be used for maximum protection. All three levels are open to the programmer, however, to provide a maximum in openness and flexibility. We will add a further, even more abstract level in Chapter 6 on page 103 where we describe the processing of design files in the encapsulation service.

## 5.4 Case studies: Implementing design information management

### 5.4.1 Approach

The conceptual schema presented in this chapter provides only the logical organization of a design information management system. It relies on a design data manager (DDM) that implements the necessary database functionality. There are basically three choices to build such a design data manager:

1. Implementing one from scratch, based on a low level storage mechanism like the UNIX file system or rudimentary database facilities offered by the operating system, like the DBM library. The advantage of this approach is the flexibility in choosing an appropriate and efficient implementation; disadvantages are high efforts for coding and maintenance, and the difficulty to share design structure information with existing DIM systems (Figure 20-A).
2. Using a general purpose, domain neutral database management system (DBMS). This approach is attractive because a robust, powerful, and fast database management system can be chosen. The complete freedom to choose an appropriate mapping between our conceptual schema and the modelling primitives offered by the DBMS makes an efficient implementation possible. Depending on the models supported by the database system, our conceptual schema may be mappable to the database with little or no effort (Figure 20-B). Of course, the capabilities of the DBMS must match our application domain:
   - **Modelling.** In order to easily map our conceptual schema to the data model the DBMS should support the concepts of *aggregation* and *inheritance*. In addition, we need support for *large objects* (byte arrays) to easily manage text chunks extracted from design files.
   - **Versioning.** Versioning can either be programmed or be directly supported as a database concept. If the DBMS already supports versioning as part of its data model, much implementation effort could be saved.
   - **Performance.** In addition to fine-grained access, the DBMS must be able to

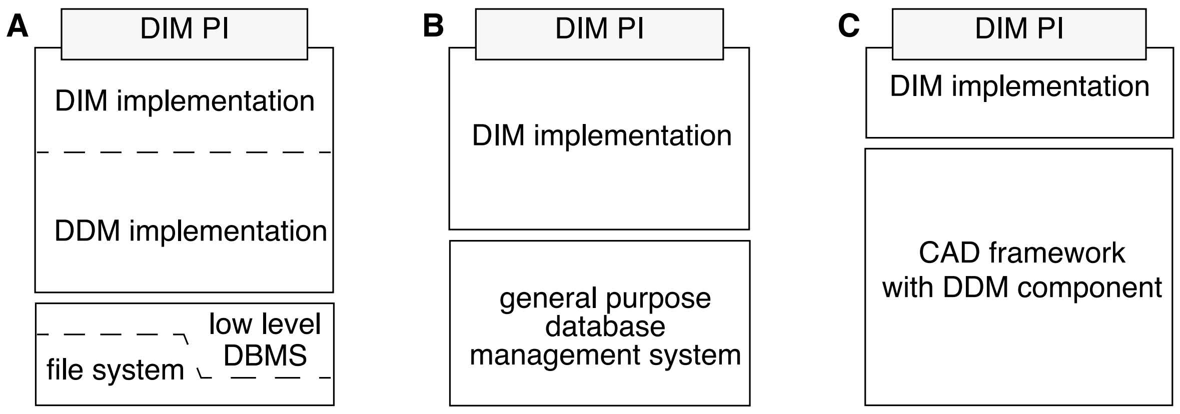


***Figure 20.*** *Three approaches to realize a design information management system.*

store and retrieve large objects with at least the performance expected from a file system. Queries are mostly needed for interactive access. Navigation of relationships between objects, however, must be fast to quickly access even large composition trees.

- **Concurrency control.** Ideally, the DBMS supports concurrency control both by *locking* as well as by allowing object *check-in/check-out* from public to private *workspaces*. A simple two-level workspace model would suffice our concept of global projects. A more advanced system supporting arbitrarily nested workspaces could conveniently be mapped to a hierarchy of libraries or projects. This would be a valuable extension to our conceptual schema.

These capabilities can be found to varying degrees in database management systems based on each of the common data models. Object-oriented systems have the richest data model in terms of modelling power. As they are tailored towards engineering applications, many of them already support versioning as part of their data model. Support for large objects no longer is a privilege of object-oriented systems. Many relational database management systems today provide a similar feature as an extension to the classical set of column types. Navigational access is inherently weak in the relational model, whereas it is the standard access method in hierarchical and network models, which, on the other hand, have weak query facilities. Finally, concurrency control is available in all database models, but only the rich type system of the object-oriented model provides the necessary flexibility in access granularity. With all this, an object-oriented database system seems to have the best combination of features for our purposes.

3. Using an existing DDM system, maybe as part of a general purpose CAD framework. This approach is attractive in that it promises the least coding effort when the schema of the DDM system is close to our schema yet open to necessary extensions. If the mapping is done carefully there is even a high probability that we will be able to integrate design management data native to the DDM system with data manipulated according with our conceptual schema (Figure 20-C).

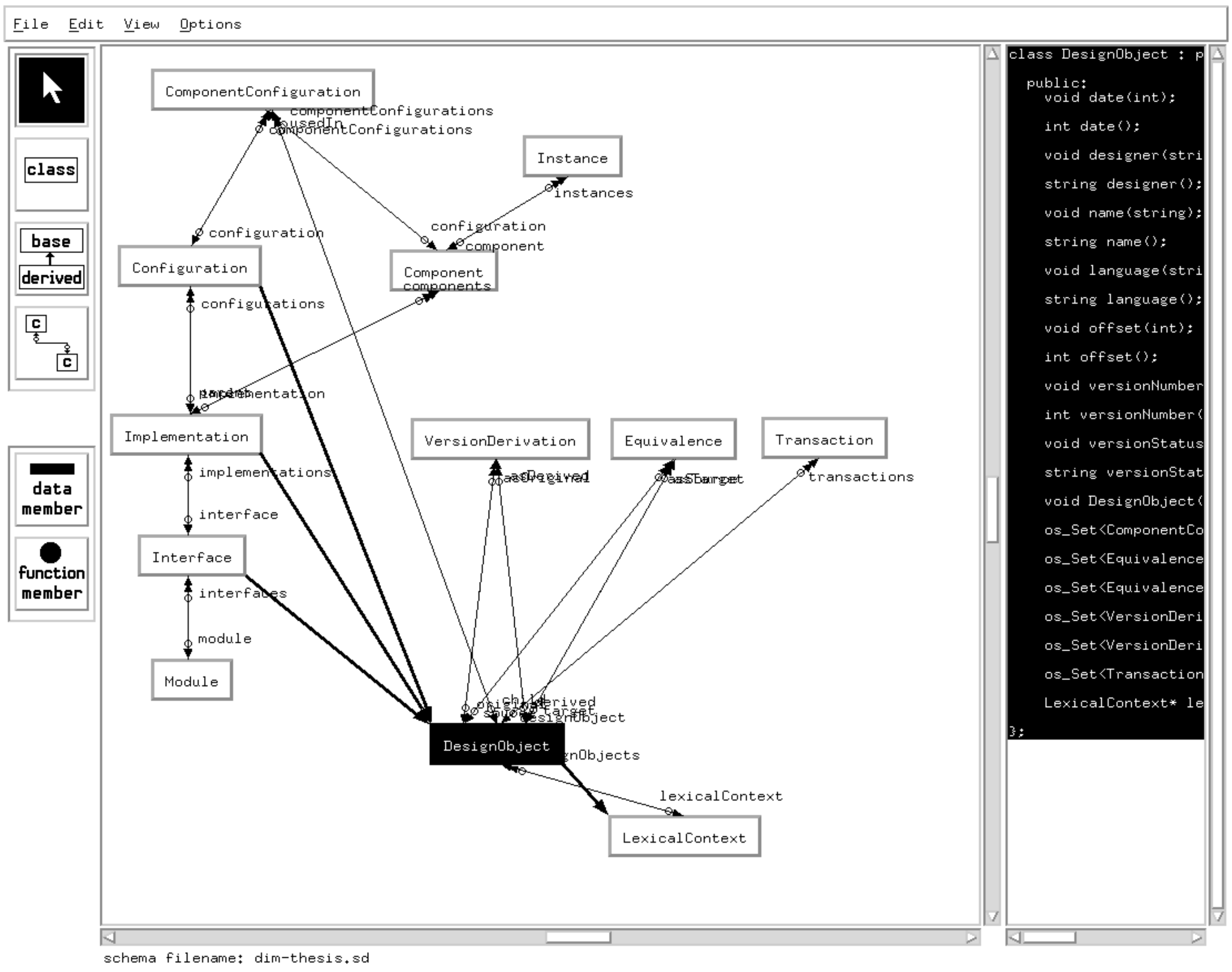
With the availability of both efficient object-oriented databases and powerful CAD frameworks alternative (1) is clearly not to be favoured. Both alternatives (2) and (3) have their specific merit and we will describe an implementation of our DIM PI using either approach in the following sections.

### 5.4.2 DIM on the ObjectStore OODBMS

ObjectStore [ObjectDesign 94] is an object-oriented database management system that allows to make "C++" objects persistent in a nearly transparent way. This is achieved by exploiting the virtual memory mapping capabilities of modern operating systems. Whenever a program tries to access the virtual memory address of an object not mapped into core memory, the memory management hardware generates a page fault. The memory page containing the memory address is loaded into the page buffer maintained by the operating system and control is returned to the program that caused the page fault, just before the malicious instruction. The instruction is executed again and now finds the addressed object in place. Memory mapping architectures of (object-oriented) databases intercept page fault signals and retrieve memory pages from a background database. On client/server architectures, this memory page may even reside in a remote machine. Persistence in ObjectStore is not a feature of a certain class, but is associated with individual objects. An object is created in a particular database or cluster within a database by passing a database or cluster handle to an overloaded version of the *new* operator.

Besides normal "C++", the programmer needs only a handful of additional procedures to

- open and close databases,
- create and access so-called database roots as named entry points into a database,
- enclose code fragments in transactions to provide protection and concurrency control.



```
class DesignObject : p
  public:
    void date(int);
    int date();
    void designer(stri
    string designer();
    void name(string);
    string name();
    void language(stri
    string language();
    void offset(int);
    int offset();
    void versionNumber
    int versionNumber(
    void versionStatus
    string versionStat
    void DesignObject(
    os_Set<ComponentCo
    os_Set<Equivalence
    os_Set<Equivalence
    os_Set<VersionDeri
    os_Set<VersionDeri
    os_Set<Transaction
    LexicalContext* le
};
```

***Figure 21.*** *Using ObjectStore's SchemaDesigner to map the DIM schema to "C++" classes. Role names are sometimes obscured due to deficiencies in the graphical user interface of the tool.*

Other than that, pointers into virtual memory are used as object references. ObjectStore provides many other features that make it an ideal basis for an engineering database, e.g.:

- nested transactions
- collections that can be indexed and queried
- version management with configurations and nested workspaces
- schema evolution

Due to the support for persistent "C++" objects, the DIM PI can be mapped directly to persistent "C++" classes (Figure 21). Set-valued attributes are implemented through ObjectStore's *collection classes*. Consistency is automatically maintained by ObjectStore by the declaration of *inverse members* so each time an object reference is set as the value of a data member of an object, ObjectStore ensures that this object gets inserted into the corresponding set-valued attribute in the referenced object.

### Accessing design structure

In our implementation on ObjectStore, each project is stored in a separate database. A project database contains a global workspace where shared objects are deposited. A call to *dim_Project::open* creates a new workspace as child of this global workspace and starts a transaction. The workspace is associated with the project handle returned by *dim_Project::open* and is used for check-in and check-out of design objects. The transaction guarantees that all changes can be rolled back when the project is closed with mode *DIM_ABORT* or when the application program crashes.

The classes *DesignObject* and *Module* are defined as sub-class of the system-defined class *os_configuration*. A configuration in ObjectStore defines a set of objects that evolve together. We use configurations together with hierarchical workspaces to implement automatic versioning. When an *os_configuration* is checked out from the global workspace into the project workspace associated with a project handle, a new version of it is created in the project workspace that is only visible there. Only on check-in this version is frozen and made visible to other users of the same project. Access by name through the functions *getModule*, *getInterface*, *getImplementation*, *getConfiguration* is supported by maintaining the class extents of the respective classes as database roots.

### Accessing design representation

Transactions on design objects are implemented by check-in and check-out of design objects into the private workspace associated with an open project. As long as design objects are manipulated in this workspace, they are not visible from the global workspace or by other of its children. The exact effect of invoking either of the functions *openInterface*, *openImplementation*, or *openConfiguration* depends on the type of design object passed as first parameter as described in Section 5.3.3.

### 5.4.3 DIM on the Nelsis CAD Framework

Implementing design information management on the Nelsis CAD Framework [Dimes 93b] is not as straightforward as on ObjectStore. This was to be expected, however, because the individual object types in our conceptual schema have to be carefully mapped to suitable types in the Nelsis schema so that as much Nelsis functionality as possible can be used. The reward of a successful implementation is that the design tools and framework tools already integrated with Nelsis can be used, making it easier to build a complete design environment. Where no suitable concept exists in Nelsis (e.g. component configurations), the implementation has to emulate these types by

- adding new types to the Nelsis schema and/or
- storing some information in streams.

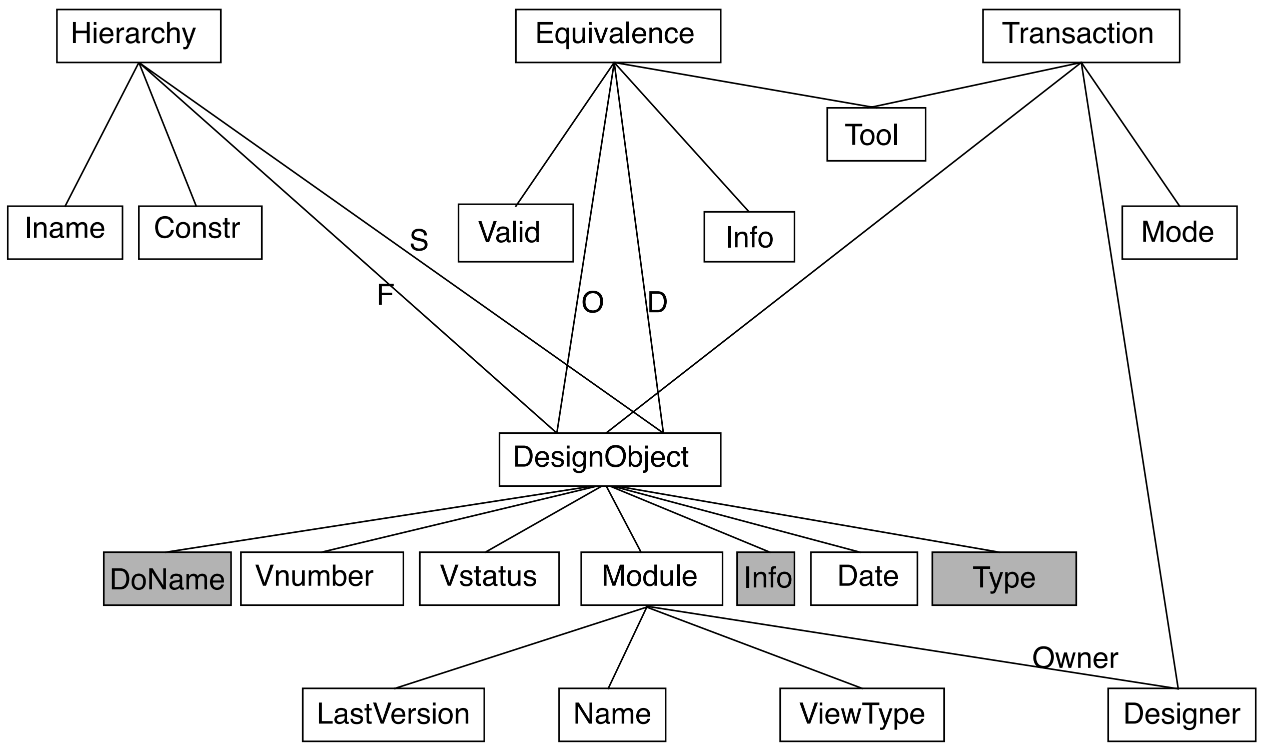


***Figure 22.*** *Mapping to Nelsis. The shaded boxes denote the added types.* Type *is used to store the actual type of a design object (interface, implementation, configuration, or context).* DoName *on* DesignObject *stores the name of a design object.* Info *stores miscellaneous information items about design objects.*

The fundamental object type in the Nelsis 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-viewtype-version-set, called *Module*. A module describes a specific aspect of an (electronic) system, enumerating different ways to achieve this aspect. It is uniquely defined by the pair *(name, view type)*.

Every design object is uniquely defined by its version number and its module. Additional attributes of design objects hold its version status (either *backup*, *actual*, *working*, or *derived*), and its modification date. A *working* version represents work-in-progress and is overridden when a new version is checked in. Only one working version may exist in a module at any one time. When the designer is confident about the maturity of the *working* version, she can freeze and publish it by changing its version status to *actual*. Nelsis assumes an *actual* version to be frozen and published. This version is used automatically for constructing composition hierarchies if no other version is explicitly selected. It is protected against overriding by the system. When a new *working* version is designated as *actual*, the old *actual* version becomes a *backup*. *Backup* versions are retained to maintain a complete design history. In addition, versions may be deliberately designated as being *derived* from another design object to protect them against overwrite. There may be many *backup* and *derived* versions in a module.

Design objects may be related by the many-to-many relationships *Hierarchy* and *Equivalence*. *Transactions* record design transactions performed on a particular design object and thus allows to retrieve its history. Equivalences are used for many purposes, one not so obvious being a version derivation relationship. Other uses include relating design objects from different view types. Here it is anticipated that the actual validation of the equivalence is performed by some tool such as a design rule checker, a circuit extractor, or a synthesis tool. We only describe the core of the complete Nelsis schema as far as it directly relates to our implementation of design information management. The production schema contains many more types for trans-project references and design flow management. A more thorough discussion of all the features of the Nelsis schema can be found in [vanderWolf 93].

## Accessing design structure

We now describe our mapping from DIM types to types in the Nelsis schema. The added types are shaded in Figure 22. The changes have been kept to a minimum, trying to stick closely to the notions inherent to Nelsis design management. The central problem in mapping the conceptual design information management schema to Nelsis is how to map the specializations of *DesignObject*. In Nelsis a design object is the only entity that is actually associated with design data.

The DIM type *Module* is mapped to a set of Nelsis *Modules* that all have the same name, but different values of the *ViewType* attribute. The function *getModule*

```
Set<Module*> getModule (string moduleName);
```

is implemented with the query

```
get Module where Name == '$moduleName'*
```

The DIM type *Interface* is mapped to the Nelsis type *Module*. As a module in Nelsis cannot be associated with design data directly, a single design object in each module is designated as the container for the interface description.

The function *getInterface*

```
Set<Interface*> getInterface (
  string moduleName, string interfaceName, string levelOfDetail);
```

is implemented with the query

```
get DesignObject where
  Module its Name == '$moduleName' and
  Name == '$interfaceName' and
  Type == 'interface'
```

*. The function parameters in this and the following queries denote variable references using a '$' imposed by using the Tcl language as extension language in our implementation.

The DIM type *Implementation* is mapped to the Nelsis type *DesignObject*. The function *getImplementation*

```
Set<Implementation*> getImplementation (
  string moduleName, string interfaceName,
  string implementationName);
```

is implemented by retrieving an interface design object with the function *getInterface*, storing its module as *module,* and then issuing the query

```
get DesignObject where
  Module == '$module' and
  Name == '$implementationName' and
  Type == 'implementation'
```

The DIM type *Configuration* is also mapped to the Nelsis type *DesignObject*. The function *getConfiguration*

```
Set<Configuration*> getConfiguration (
  string moduleName, string interfaceName,
  string implementationName, string configurationName);
```

is implemented similar to *getImplementation* by retrieving an interface design object with the function *getInterface*, storing its module as *module*, and then issuing the query

```
get DesignObject where
  Module == '$module' and
  Name == '$implementationName' and
  Info == '$configurationName' and
  Type == 'configuration'
```

The DIM type *LexicalContext* has to be modelled as Nelsis type *DesignObject* because it is associated with design data. For each lexical context a new module and in it a single design object is created. The module has view type '$language-context'. Syntactical contexts are not mapped to Nelsis. Instead, they are dynamically constructed on export from a parse tree template and attribute values of the participating design objects.

The other types of the DIM schema are mapped as follows. *Instances* are not represented at all. For the export of design objects as files it is irrelevant how many and which exact instances are used by a particular implementation as long as the child design object to be exported is known. *ComponentConfigurations* are identified with the *Hierarchy* objects a design object that implements a configuration is involved in. *VersionDerivation* is handled automatically by Nelsis and is mapped to Nelsis *Equivalence* relationships. *Transaction* is also handled automatically.

## Accessing design representation

The function *dim_Project::createInterface*

```
dim_Interface* createInterface (
  string moduleName, string interfaceName, string language,
  string levelOfDetail);
```

first tries to create a new module using the query

```
insert Module its
  Name = '$moduleName',
  Designer = '$env(USER)',
  ViewType = '$language-$levelOfDetail'
```

The variable *language* here denotes the hardware description language used for the interface. The variable *env* is a global associative array in Tcl that gives access to environment variables. An error is signalled when such a module already exists. Then, a design object *do* in this module is opened for update. As Nelsis does not know about the additional attributes on *DesignObject* in the Nelsis schema, they have to be set manually:

```
update DesignObject '$do' its
  Name = '$interfaceName',
  Info = '$language',
  Type = 'interface'
```

Note that with this mapping approach, interfaces cannot be versioned. Versioning on interfaces has to be simulated by creating them in separate DIM modules. The function *dim_Project::createImplementation*

```
dim_Implementation* createImplementation (
  string moduleName, string interfaceName, string implementationName,
  string language, string domain);
```

inserts a new implementation design object into the module that contains the associated interface object. Versioning is done automatically by Nelsis. The additional attributes again have to be updated manually:

```
update DesignObject '$do' its
  Name = '$implementationName',
  Info = '$language',
  Type = 'implementation'
```

An implementation design object also serves as its own first configuration. This accounts for the fact that Nelsis does not support explicit configurations. The function *dim_Project::createConfiguration*

```
dim_Configuration* createConfiguration (
  string moduleName, string interfaceName,
  string implementationName, string configName);
```

works similar to *dim_Project::createImplementation*. The additional attributes are updated as follows:

```
update DesignObject '$do' its
  Name = '$implementationName',
  Info = 'configurationName',
  Type = 'configuration'
```

The function *dim_Configuration::getInstances*

```
Set<dim_InstanceInfo*> dim_Configuration::getInstances
  (dim_BindingMode bindingMode);
```

with binding mode *dynamic* simply issues the query

```
get Hierarchy its Name, Constructor, Son-DesignObject,
  Son-DesignObject its Type
```

to create a set of instance information structures. For binding mode *static,* the function starts off with the same query. According to the type of the child design object, the appropriate implementation design objects have to be selected as described in Section 5.3.

### Discussion

The approach to a mapping between the DIM conceptual schema and PI to the Nelsis schema and associated queries described above is certainly not the only viable solution. Another way to map interfaces and implementations to types in Nelsis 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 this approach, 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 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 separately. 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.

Using this approach, a module in Nelsis does not represent a single interface but rather a set of implementations with similar interfaces (Figure 23). Implementations with incompatible interfaces are collected in separate modules, related to each other by equivalence relationships. While this approach is quite valid on its own, it does not fit equally well into the Nelsis philosophy of identifying a module by its name and view type. Many dissimilar design objects are collected within a single module. It is also not possible to support an *actual* version for each interface as Nelsis only allows *one actual* design object per module.

## 5.5 Related work

The work presented in this chapter originated from the need to manage structural design information and its attached design representation according to the requirements stated in Section 1.6 on page 15. In addition, the anticipated solution had to be formally defined by an information model and should be both flexible and natural to the designer. It should deviate as little as possible from existing work in the area of design data management using CAD frameworks. To our knowledge, no system fulfilling all these requirements has been described in the literature.

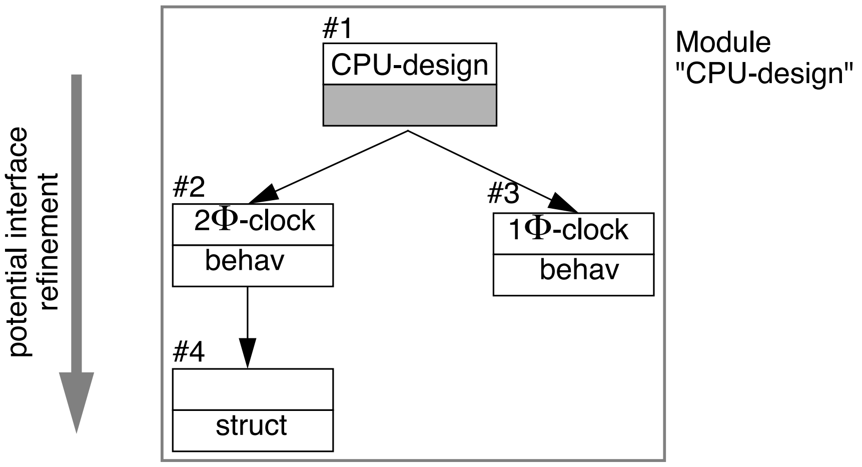


***Figure 23.*** *An alternative mapping solution: Implementations with similar interfaces are represented by a module. The box at the root of the tree has no implementation and is the target for dynamic hierarchy relationships. Arrows denote version derivation. The figures are consecutive version numbers.*

A configuration management approach more powerful than the one found in e.g. the Nelsis CAD Framework was called for by the need to represent VHDL configurations in the DIM service. Also, Nelsis does not distinguish between interface and implementation, a prerequisite for top-down design.

In [CFI-FAR 93], CFI outlines the concept of references that are bound by configurations. Although this document does not describe an existing, working system, the concepts described form the basis for existing frameworks such as the JESSI Common Framework. We have transcribed the intuitive drawings from [CFI-FAR 93], "Data Management Information Model" and "Version Information Model", into the Xplain notation to be able to compare it to our conceptual schema (Figure 24). This transcription can only serve as a coarse approximation of the original intent of the authors of the CFI document. Although no cardinalities were attached to relationships in the informal CFI schema, the appearance of an Xplain schema is largely determined by the different cardinalities so cardinalities had to be guessed from the textual description. Also, type *Binding* was only described textually.

The first observation is that this schema does not distinguish interface and implementation of a design object from each other. This may be due to the fact that more domain specific schemas are intended to augment this schema with domain specific specializations of the types defined here. The overall structure of the types *Configuration/Binding/Reference* seems to resemble our types *Configuration/Component-*

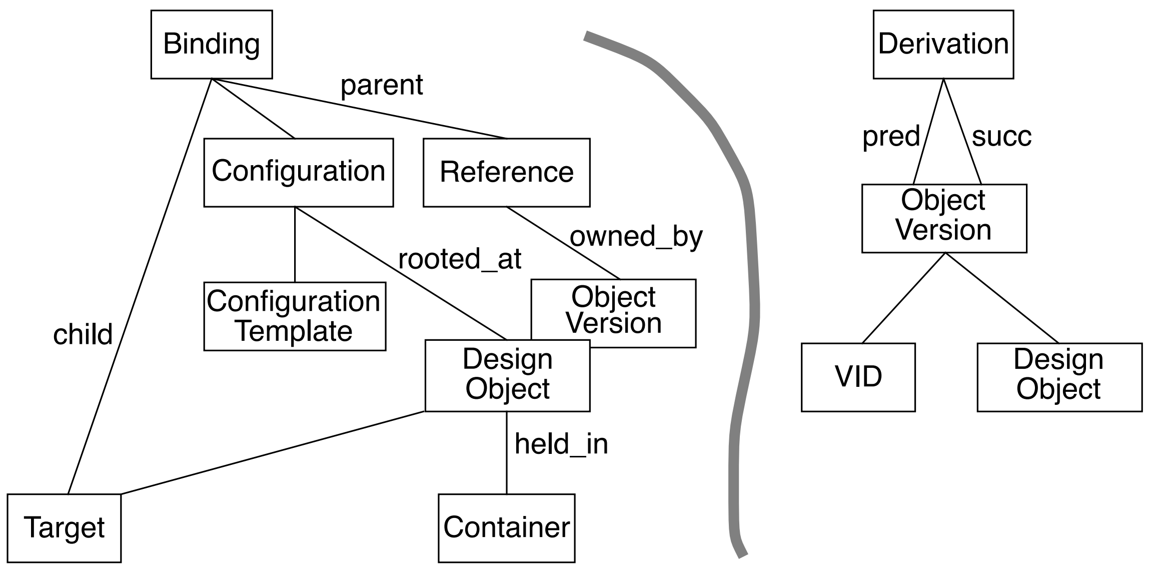


***Figure 24.** Transcription of CFI's "Data Management" and "Version" information models into Xplain.*

*Configuration/Component* as depicted in Figure 13 on page 72. However, CFI does not restrict their retargetable *Reference* to composition hierarchies. The binding policy we have implemented globally in the programming interface (see Section 5.3.4 on page 87) is bound to the type *ConfigurationTemplate* in the CFI schema so that in principle different binding strategies can be used. We have no information as to whether such a system has been implemented yet.

# 6. Processing design description files

## 6.1 Overview

In accordance with requirements **R6** and **R7** and the integration principle (2), stated on page 40, we assume that design information is exchanged between encapsulated design tools and the design information management (DIM) service of an EDA framework by means of design files. In the EDA domain, hardware description languages (HDL) are used to encode design information and unambiguously define design semantics. Due to the many aspects of electronic design (domains, levels of detail) and varying design tool requirements, there are many HDLs in frequent use. Some languages cover many aspects of a design, others are specific to a single domain, level of detail or even design tool. Recent years, however, have seen a strong trend towards HDL standardization as only with standardized HDL syntax and semantics there can be seamless interoperability between design tools in an open, integrated design environment. Some design description languages and their preferred area of use are listed in Table 10.

| language | mostly used for ... |
|---|---|
| VHDL | behavioural and structural descriptions on algorithm and functional block levels |
| Verilog | behavioural and structural descriptions on register transfer and gate levels |
| EDIF | netlists and graphical data for schematics and physical layout |
| GDS-II | graphical data for physical layout |
| C++ | behavioural descriptions on algorithm level |
| Prolog | behavioural descriptions on architecture and algorithm levels |

***Table 10.*** *Some design description languages and their preferred area of use. Note that we have included two ordinary programming languages which are frequently used for behavioural descriptions on low levels of detail.*

This diversity of languages in design use implies that vendors of EDA frameworks make an arbitrary selection of the languages they want to support, mainly based on customer demands. As design methodologies are different from company to company and also evolve constantly, a framework vendor continuously has to support new design description languages. He may quickly be in a position in which supporting new languages[*] based on customer demands forms a major part of the overall effort put into framework development. There are three viable approaches to solve this problem:

1. Not to support design description languages at all. The JESSI Common Framework, for example, never looks into a design file but relies on the designer to chop his designs in pieces small enough to be managed individually by the DIM service of the framework.
2. To make an arbitrary selection of the languages to be supported. All design tools integrated with such a framework (e.g. Nelsis) have at least to provide converters from their native languages to the languages supported by the framework. If the supported language selection is not appropriate, and unsupported languages have to be used by design tools because a certain domain or level of detail cannot be represented, design files have to be manipulated manually by the designer to be digested by the framework.
3. To provide a framework service that allows a framework administrator or even design team members to add support for a desired design description language on the fly. Of course support for standard languages can already be provided by the framework vendor. Incorporating support for specific design description languages into an existing framework can even form the basis for a flourishing business niche.

Clearly, the third alternative is the most flexible approach. Without special support, however, the effort of writing the necessary language processors for complex languages like VHDL is a major engineering task and certainly out of the scope of a design team wanting to design chips.

[*]. "to support a design description language" here means to be able to manage structural information contained in design files by the DIM service and by framework tools like query interfaces and browsers.

Traditionally, writing language processors is accomplished by any of the following alternatives:

1. combination of the UNIX tools *sed/awk*, or dedicated report generation languages like *perl*[*]
2. *lex* generated lexical scanner
3. *yacc* generated parser
4. *yacc* generated parser enhanced with semantic actions to resolve references

Using (1) requires little coding effort and development of language processors can be incremental due to the interpreting nature of the tools applied; the resulting language processors, however, have only crude recognition capabilities and high probability of failure due to syntactic variations in the processed design files. Alternative (2) requires the compilation of generated “C” code; it provides more flexibility in the actions that can be triggered when recognizing a language construct. Language specification still is based on regular expressions and therefore too crude for most real design description languages. Alternatives (3) and (4) have detailed, syntactical language recognition capabilities. The coding effort is high because every language detail must be processed. The language processors thus created are highly dependent on fine-grained detail of the processed language. As design tools often rely on *dialects* of a standard language, this is an undesirable feature, even more so as the processing of every single detail of a design description is not necessary for successful tool encapsulation.

The choice of standard compiler construction tools like *yacc* and *lex* suggests that the processing of design files is very similar to the initial parsing phase of a compiler for a particular design description language. There are, however, significant differences:

- Lexical and syntactical analysis in a compiler try to digest every single detail of a design description to be able to derive an internal representation that matches the information content of the design description as thoroughly as possible. For the purpose of design tool encapsulation, only a small part of design files has to be analysed down to single lexems. Large parts can simply be skipped and regarded as design representation, uninterpreted as far as design information management is concerned.

[*]. Perl gains more and more acceptance as one of the primary scripting languages in EDA environments (cf. [CFI-EII 95])

- A large part of the lexical and syntactical analysis phases of a compiler is concerned with error detection and recovery. As this tedious work is performed by design tools already, we can rely on a design description to be syntactically correct for the purpose of tool encapsulation.
- Another great effort in the initial phase of a compiler is the construction and maintenance of a symbol table for semantic analysis. We are not interested in the exact data or control flow and can therefore greatly reduce the effort of symbol handling. The close cooperation of the new encapsulation service with a DIM framework service with versatile querying facilities renders manually constructing a symbol table obsolete.

On the other hand, there are also requirements in the realm of tool encapsulation that are not even an issue with ordinary compiler technology:

- All text that is considered design representation does not have to be interpreted in any way but must be saved complete for later perusal by design tools. Once an ordinary compiler has built an internal representation of the input read, it can safely forget about the exact textual representation, with the possible exception of line numbers for error messages.
- Global text that neither belongs to structural information nor is associated with single chunks of design representation must be preserved to be able to reconstruct complete design files later. In a compiler, this text is sometimes already resolved by a preprocessor and not even seen by the actual compiler or it serves as context information and is also translated to objects in an internal representation.

We conclude that an EDA framework has to offer a powerful and flexible service to incorporate support for specific design description languages. While this service certainly should provide the language processing power of conventional compiler construction tools, additional requirements have to be met. To understand the implications of these requirements, we now take a closer look at which kind of language processing is actually required to support a selected design description language in an EDA framework.

## 6.2 Requirements

### 6.2.1 Design file import

Design files can either be created manually by a designer or can be the result of a design tool execution. Whereas in the former case the designer would be free to arrange the files that describe a complete electronic system in any way that is required for design information management, design tools generally do not provide this flexibility. This implies that design files have to be analysed to extract the structural information necessary to perform design information management. Corresponding to the notions defined in our conceptual schema the following items have to be created:

- interface, implementation, and configuration *objects*
- *attribute* values for names, designer, modification dates
- *relationships* for compositional hierarchy, version derivation, and equivalence

In addition, textual design representation has to be associated with objects so that it can be retrieved again on export.

### 6.2.2 Design file export

When performing a design step by running a tool, design information must be exported from the DIM service to design files. This can be achieved by traversing the graph of objects managed by the DIM service according to our conceptual schema, starting from a selected root object, and retrieving the design representation that is associated with the objects visited.

The existence of component configurations in a design hierarchy complicates matters. In cases where a design description language is used that supports configurations (like VHDL), the configurations managed by the DIM service can be translated into appropriate configuration declarations in the language and emitted along with the selected interfaces and implementations. If, however, the design language does not have a notion of “configuration” or the notion supported by the language grossly deviates from the notion used by the DIM service, configurations have to be resolved and emitted as static bindings between compound implementations and their components.

### 6.2.3 Associating binary objects with structural information

The import and export mechanisms described above assume that design files have a well-defined structure and contain some kind of parseable and printable description of the design. Quite often, however, design tools produce and read an opaque, intermediate design description. Examples for this kind of files 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 hierarchy they can 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 have to be imported and exported whenever a sub-module of this root object is imported or exported. We now introduce our approach to providing a framework service that fulfils these requirements.

## 6.3 Approach

We have developed a toolkit that can be offered as a new service by framework vendors to their customers or can be used by framework vendors to build design language support for standard languages into their products. We follow the principle of separating syntax specification from the actual creation of objects on import and retrieval of design information on export. The syntax specification of a given HDL is largely static and parsing can be greatly simplified when using standard compiler construction tools like yacc and lex. The actual interfacing to a DIM service is likely to evolve as requirements on design information management in a particular design situation mature. This part of language processing can therefore benefit from the accessibility gained from coding in an interpreted extension language. As suggested in Section 4.3, we use an extended Tcl for this purpose.

### 6.3.1 Syntax specification

As opposed to the situation in an ordinary compiler in which a program in some high-level programming language is translated into a program in a different language like assembler or machine language, input and output to and from the DIM service is in the same language.* As such, it is desirable to use the same syntax specification for both import and export.

As stated earlier, we can assume that only little information out of the whole information content of a design file is needed for successful design information management. We can furthermore assume design files to be syntactically correct. These two assumptions greatly simplify the amount of syntax that actually has to be specified exactly. On the other hand, partial analysis poses additional problems not normally encountered in syntax analysis. While large parts of a design file can be analysed according to a lexical and syntactical specification that generously ignores most of the tedious detail of a complete design description, other parts have to be analysed down to single lexems in order to retrieve all the information necessary for design information management. For example, the analysis of a VHDL file only has to reveal the names and extent of contained entity declarations and architecture bodies, but *all* the information specified in a configuration declaration has to be retrieved.

We solve this problem by allowing modular language specifications. The specification of a design description language may consist of several modules, each of which defines the lexical properties and syntax for a part of the whole language. There is always a main module that provides an entry point for syntax analysis and generation. With the recognition of certain constructs, another module may be invoked that takes over syntax analysis from where it was invoked. Once a complete program defined by the language of the subordinate module has been recognized, control is returned to the caller. As different modules can define completely different lexical properties, modular language specifications allow to analyse design files at varying granularities. The net result is a major reduction in the amount of language detail that has to be specified for design file analysis while we are still able to digest fine-grained detail where appropriate.

We use the same language specification to define both input and output. As such, the specification language needs to define both lexical properties and syntax in one place. In addition, grammar symbols may be tagged for ease of reference during import and export. The specification may be decorated with output formatting information.

*. We regard situations in which design information has to be converted from one language to another as the task of dedicated design tools.

### 6.3.2 Parse tree construction

Once analysed, the design information retrieved from design files must be organized in a structure that is easily traversed from extension language scripts to create the corresponding objects, attribute values, and relations in the DIM service. A parse tree can be created automatically during syntax analysis, and, combined with an extension language binding, is well suited as intermediate representation. We choose a generic data structure for the parse tree that is independent of the actual language syntax.[*] Parse trees are constructed from generic nodes that carry type information, user defined tags, and text that stems from analysed designed files.

We provide no operations to prune the automatically created parse trees, because there is little need for this extra complexity. With modular language specifications, parse trees proved to be small compared to those created from complete fine-grained syntax analysis. Parse trees directly reflect the extended BNF by which the syntax is specified. Tree nodes that stem from certain syntax rules carry the name of this rule as their type. Lists, possibly with separators like "," or ";", create dedicated list nodes. Text ranges that are parsed by nested specification modules result in sub-trees.

### 6.3.3 The creating of objects during parse tree traversal

The parse tree created automatically during syntax analysis has to be traversed to determine the objects, attribute values and relationships to be created in design information management. While all the data from the analysed design files is guaranteed to be in the parse tree, it is obscured by intermediate nodes that stem from the structure of the input grammar. We define traversal operations that access child nodes based on their type and optional tag values, ignoring intermediate tree nodes that do not carry valuable information. These traversal operations are bound to Tcl commands, so all tree processing can be done by extension language scripting. While visiting tree nodes, the programmer can store values found in variables and use their values to fill the attributes of objects in design information management once enough information is gathered.

[*]. Another popular approach for parse tree construction is the definition of special node types for individual grammar rules. While this seems attractive from a purist's point of view, it would require to define a subtype of the basic node type for each grammar rule. On the other hand, modules with well-defined interfaces rather than deep inheritance trees have proven more usable in practice [Udell 94].

We prefer the programmed approach to the more descriptive approach of statically linking language constructs to DIM object types because thus

- the relatively fixed language definition can be separated more cleanly from the mapping to DIM object types that is likely to evolve as information management requirements change;
- more flexibility can be offered to the integrator to gather values from anywhere in the parse tree. Otherwise, a complicated formalism would be necessary that would yet be likely to fail in unexpected situations.

### 6.3.4 Linking types to parse trees

Export from the DIM service to design files is an altogether different matter. Here we start from a well-structured object graph that already has all the information needed and is attached to objects in the DIM service so that it is easy to access. For every language specification module, we automatically generate a function *unparse* that takes the name of a syntax rule to be unparsed as parameter as well as a variable number of parameters that match variable components in a derivation of the named rule. DIM object types are statically associated with an invocation of the *unparse* function that determines the syntax rule according to which objects of this type are to be exported, as well as the association of attribute values to the rule's variables.

With this association in place, export of a DIM object into a valid design file is a matter of selecting the object and invoking its unparse method. The unparse method creates a parse tree with constant text in the right places and variables filled from the DIM database. Only a simple preorder traversal is needed to transform such a tree into a stream of text in the desired syntax.

Now that we have outlined our approach, we will present the technical realization of this new framework service in Sections 6.4 through 6.6, use VHDL as a case study in Section 6.7 and compare our approach with existing work in Section 6.8.

## 6.4 A specification language for HDL syntax

Our specification language has to specify the syntax of a design description language for both import and export. As such it allows to define both lexical properties and syntax in a single specification file. We have already described the basic language in Section 3.2, so here we only add the constructs that are specific to encapsulation support.

As a rule, the generated parser modules assume that they process syntactically correct design files. They therefore read over lexems that are not explicitly defined in the language specification. This behaviour can be exploited to simplify the language specification: Only those languages elements have to be defined that either carry valuable information for design information management or those that are necessary to recognize the syntactical structure of a design description. All other text will be ignored by syntax analysis but will nevertheless appear as text attributes in the parse tree to ensure that no information is lost for later export.

### 6.4.1 Nested parser modules

As specified in Section 3.2, the main part of a language specification consists of a list of grammar rules. A rule may be marked with the property *-syntax*. This property marks the right-hand side of a rule to be the prefix for a nested specification module. Once the prefix is recognized, it is pushed back into the input and a subordinate parser module with the same name as the rule's is used to specify the input that follows.

When the subordinate parser has recognized a valid text in the language that it defines, it returns the control back to the invoking parser.

```
rule rule {
   "rule" ID opt { "-syntax" } alternatives
}
```

### 6.4.2 Generating a parser from specifications

Figure 25 illustrates the process of generating a parser module from a language specification. A parser module provides the two functions *<module>parse* and *<module>unparse* as its public interface, and a wrapper function to create a parse tree from an extension language command.

There are two variants of the parse function:

```
Tree* parse (char* fileName);
Tree* parse (Tree* parent);
```

While the first variant is invoked directly from the extension language binding to create a parse tree of the named file, the second variant is used to pass control to a subordinate parsing module. The unparse function has the following signature:

```
Tree* unparse (char* rule, ...);
```

The first parameter always denotes the syntax rule to be unparsed. The following parameters supply name/value pairs for variable parts in a derivation of the named rule. We will explain the wrapper function to create parse trees when we describe the extension language binding to parser modules.

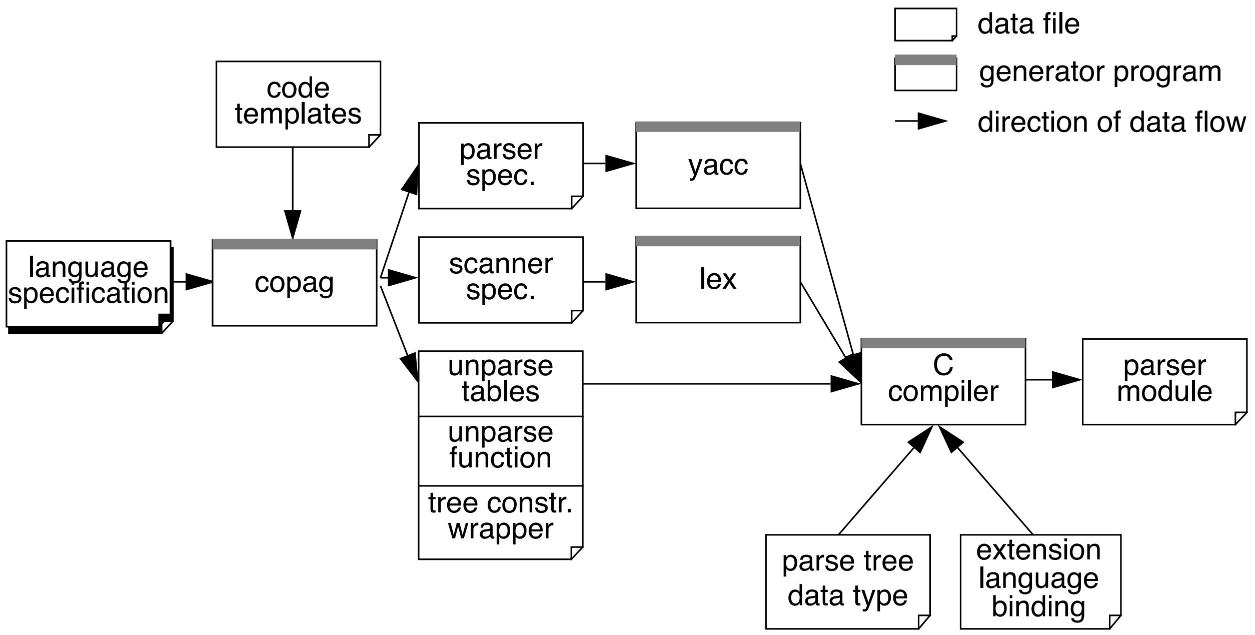


***Figure 25.*** *How to generate a parser module from a syntax specification. The preprocessor* copag *reads one or more specification modules and translates them into a parser specification in* yacc *format, a scanner specification in* lex *format, unparse tables, an unparse function that uses these tables, and a wrapper function for the extension language binding of language specific parse tree constructors. The parse tree abstract data type itself is language independent and is simply linked into the created module together with a set of wrapper functions for extension language bindings.*

## 6.5 The generated parser and tree constructor

### 6.5.1 A conceptual schema for parse trees

We have chosen to construct parse trees from generic nodes that are independent of a particular language syntax (Figure 26). Each tree node is associated with a parse tree. A parse tree not only represents the root of a tree but carries information common to all the nodes in it. An important property of our parse trees is that the tree root is associated with all the input text recognized, including all text read over by the parser. This is necessary because regardless of the (in-) completeness by which the language specification defines the lexical properties and syntax of a design file, the whole file is considered syntactically correct and therefore contributes to a complete design description. All text is to be managed by the DIM service. To achieve this, we do not read a design file byte by byte with stream-i/o functions but use the virtual memory management facility offered by all modern operating systems to map the complete design file into virtual memory. The individual tree nodes do not store copies of the lexems they were associated with during parsing. They rather store the start and end offsets of their associated text regions in the mapped file text.

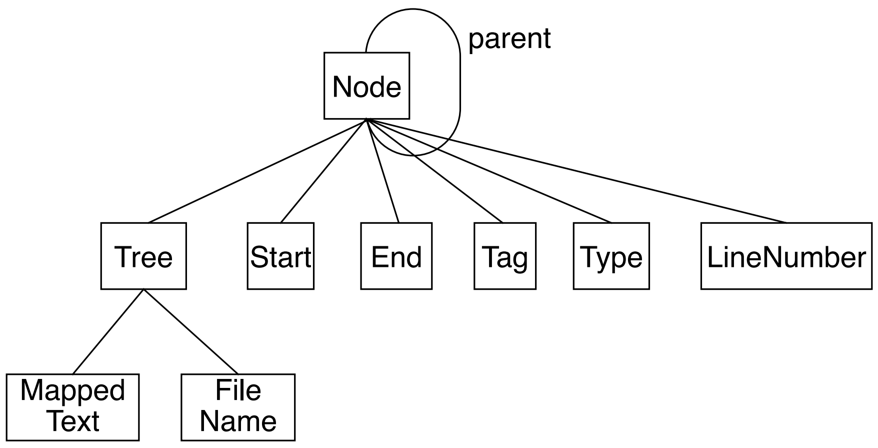


***Figure 26.*** *Schema for parse trees. An analysed design file is mapped into virtual memory and is associated with the tree. Tree nodes only carry the start and end offsets into this mapped text.*

### 6.5.2 Creating a parse tree

The creation of parse tree nodes during parsing is straightforward. A leaf node is constructed when a lexem is recognized as token in the input. Its start and end values are determined by the respective start and end offsets in the input file. Intermediate tree nodes are created when a grammar rule is reduced. They receive the start value of their left-most child as start and the end value of their right-most child as end value. Thus all the text lying in between the left-most and the right-most children of a tree node is considered to belong to this node, even if there is no child that actually represents this text.

### 6.5.3 Nested parser modules

Grammar rules that have the *-syntax* property do not directly create a tree node when they are recognized, but delegate its construction to a subordinate parser module. For this purpose the parse procedure of the desired parser module is invoked with the current parse tree as parameter. When the subordinate parser has successfully constructed a parse tree for a region of input text, it returns control to the semantic action of the invoking parser module. There the tree nodes resulting from the right-hand side of the current grammar rule are discarded and replaced by the root of the parse tree created by the subordinate parser.

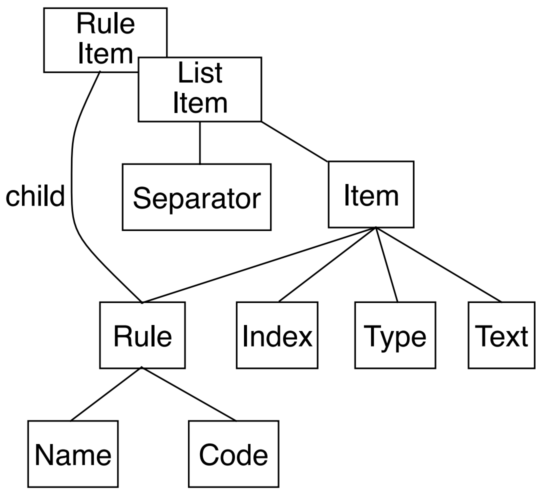


***Figure 27.*** *Schema for the unparse tables. Rules carry their name and a numeric code. An item may either be a literal or optional (both stored as object of type* Item*), or it may be a reference to a subordinate rule. List items carry a separator if one was specified in the language specification.*

### 6.5.4 Unparsing

We have chosen to generate one *unparse* function per parser module that receives its knowledge about grammar items from two tables, *RuleTable* and *ItemTable* (Figure 27). These tables can be statically created during the run of the specification preprocessor *copag*. An alternative solution would have been to generate one function per grammar rule. We favour the table driven approach because it reduces code size and has less code duplication. The *unparse* function starts from a rule whose name is passed as parameter and creates a parse tree node for it. The function then iterates over the items in the rule. For each item, a child of this parse tree node is created by the following actions:

- If the item is a literal, the created child takes the literal text as value of its *text* attribute.
- For all other item types, the unparse function first checks if a named parameter has been given to the function that matches this item. A named parameter matches an item if it is yet unused and
  - if the item is tagged and the parameter name equals the tag, or
  - if the item is a *RuleItem* and the parameter name equals the name of the referenced rule.

  For maximum flexibility, a child node can be created from matching parameters in one of the following ways:
  - the parameter text is taken as the node text.
  - the parameter denotes a tree created elsewhere which is taken as the child.
  - the parameter denotes a list of DIM objects. A new tree node is created as child. This child receives the result of evaluating the unparse method on each object in the list as children.
- If no matching parameter could be found, two cases are distinguished:
  - the item denotes a list of one or more elements. In this case no child node can be created automatically. The *unparse* functions fails.
  - the item is of type *RuleItem*. The unparse function calls itself recursively, passing the rule and the named parameters along. The result of this recursive call is taken as the new child node.

The overall result of a successful invocation of *unparse* is a parse tree that is created according to the syntax of the rule passed to the function with all the variables filled by parameters passed to *unparse*. A simple preorder traversal writes this tree into a design file. Each node visited during this traversal is checked as to whether its *text* attribute is set. If so, the text is emitted to the file and traversal proceeds with the next sibling. If the *text* is not set, traversal proceeds with the children of the current node. We will present an example of unparsing in our case study of the VHDL language.

## 6.6 Extension language interface

### 6.6.1 Parsing design descriptions

Parse trees are represented by the abstract class *ParseTree* in the extension language.[*] The only method that *ParseTree* offers is the one to retrieve the tree root. For every parser module linked into the extension language engine there is a subclass of type *ParseTree*, named after the language specification from which this module was created. For example, our case study for the VHDL language comprises of two specifications, one for the overall structure of VHDL files, named *vhdl*, the other one for configuration declarations, named *config*. We therefore have two subclasses of *ParseTree* in this example, named *vhdl* and *config*. Each parser module contains a *parse* function that takes as parameter the name of a design file to be parsed and returns a parse tree after a successful parse. These *parse* functions are accessed through the constructors of the corresponding subclasses of *ParseTree*. For example, we create a parse tree from the VHDL file dp32.vhdl by issuing the Tcl command

```
vhdl t -file dp32.vhdl
```

This command creates a parse tree *t*. When an error is encountered during parsing, the constructor fails and no parse tree is created. In this case an error message is returned as the result of the command. The root node of this tree can be retrieved by issuing the method *root* to the parse tree:

```
set root [t root]
```

This command assigns the identifier of the tree root to the Tcl variable *root*. Nodes have three data members visible from the extension language: *type*, *tag*, and *text*. *Type* contains an enumeration value that encodes the rule from which this node was created. The domain of this enumeration is defined by the tree to which a particular

[*]. “Abstract class” means that the user cannot create objects of this class but only of subclasses.

node belongs. For example, the code 24 is assigned to nodes of type *architecture*, created by the *vhdl* parser. The code 24 corresponds to node type *block_configuration* for nodes created by the *config* parser. *Tag* contains the tag from a grammar symbol if any was defined. For example, the *config* parser assigns the tag *entityName* to the node created for the third identifier of *architecture_body* nodes. Finally, *text* contains the text region between the start and end offsets stored for a particular node. For the root node of a parse tree, this is always the complete input.

### 6.6.2 Tree traversal

Once a design file is successfully parsed and a parse tree is constructed, the information relevant to design information management needs to be extracted from the tree. For this purpose two methods are defined on tree nodes that allow flexible tree queries and traversal from within extension language scripts. As the tree is built during language parsing by automatically generated code, it contains much detail that is irrelevant to information extraction. The extension language statements take this into account by allowing to look at "interesting" nodes only. Two statements are defined:

```
rule traversal {
   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 nodes 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 or 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 *node:IDENTIFIER* or fails if no such node exists.

The method on tree nodes that implements these commands realizes the following algorithm:

***Algorithm 1. Node::traverse***

| | | |
|---|---|---|
| in | set<tags> | tags to be matched against node tag |
| in | set<types> | types to be matched against node type |
| in | variable name | Tcl variable to which to assign the node identifier currently processed |
| in | action code | Tcl code to evaluate for a node with matching tag and type |
| in | one | *one* or *all* command |
| out | success/failure | return code resulting from evaluating the action code |

```
if (node type and node tag match inputs) {
 set Tcl variables for current node type,
       current node tag, and current node text
 set Tcl variable named by parameter to current node identifier
 evaluate action code passed as input
 if (return code from action evaluation == OK) {
       if (one) {
             return BREAK;
       } else {
             return (return code from action evaluation);
       }
 }
 if (node type is list or node has children) {
       for each list element or child {
             traverse recursively,
             passing input tags and types, variable name,
             action code and one flag
```

```
            switch (return code from recursive invocation) {
            case OK or CONTINUE:
                break:
            case BREAK:
                return BREAK;
            case RETURN:
                if (node type is list) {
                    insert returned node identifier at current list position
                } else {
                    replace current child with returned node identifier
                }
                break;
            default:
                return (return code from recursive invocation);
            }
        }
    }
```

*Node::traverse* effectively performs a preorder tree traversal, checking each node visited for matching tag and type. If a matching node is encountered, some Tcl variables are set to establish the execution context for the action code, and the action code is evaluated. Depending on the return code of this evaluation, traversal either continues with the children or list elements of the matching node, or breaks off completely and continues with the next sibling of the matching node. By issuing appropriate *break*, *continue*, and *return* statements within the action code, and by recursively calling traversal commands from within the action code, the programmer is given fine control over the exact tree traversal. In the Section 6.7, we will see that this flexibility is in fact needed when we show in detail how to transform VHDL configuration declarations into DIM object graphs.

### 6.6.3 Unparsing

*Unparse* functions are defined globally in the scope of their parser module. As described above, *unparse* functions determine the parameters they actually consume during a traversal of the rule and item tables.

The following extension language command is defined:

```
rule unparse {
   module:IDENTIFIER "::unparse" "-rule" rule:IDENTIFIER
   list { "-" IDENTIFIER "{" parameter_value "}" }
}
rule parameter_value {
   "-text" LITERAL
  | "-code" tcl_code
  | "-node" node:IDENTIFIER
  | "-list" tcl_code
}
```

An *unparse* command is invoked for a specific rule. Tree node creation is controlled by a keyword that specifies a parameter to be either

- plain text,
- Tcl code to be evaluated to get some text as result,
- a tree node identifier, or
- a piece of Tcl code that - evaluated - results in a list of DIM object references. A list of tree nodes is yielded from this list by evaluating the *unparse* method for each of these objects.

For example, a block configuration in our VHDL language specification is defined as follows:

```
rule block_configuration {
 "for" implName:IDENTIFIER
      list { use_clause }
      items:repeat { configuration_item }
 "end" "for" ";"
}
```

A sequence of extension language commands that writes a block configuration to standard output would be

```
set tree [config::unparse -rule block_configuration
 -implName { -code $parent_DesignObject Name }
 -items { -list set ComponentConfigurations } ]
$tree export stdout
```

The *unparse* function defined in the *config* parse module is invoked. The implementation name is found by evaluating the method *Name* on the object stored in variable *parent_DesignObject*. Component configurations are filled in by evaluating the *unparse* methods on all the objects stored in the variable *ComponentConfigurations*. Note that no parameter has to be given for the list of *use_clauses* as this list may have zero or more elements.

## 6.7 Case study: VHDL

### 6.7.1 The language

VHDL is a mixed level design description language originally conceived as simulator input language. As such its semantics as defined in the VHDL language reference manual [VHDL 87] is defined in terms of simulation behaviour. As top-down design gains importance however, synthesizable subsets have been defined and implemented as input language for synthesis tools. VHDL supports design description in behavioural and structural domains.

VHDL provides a good case study of our specification language for the following reasons:

- VHDL is important. Almost all ECAD vendors support VHDL input to their design tools, be it high-level synthesis systems, mixed-level simulators from algorithmic down to gate-level, or even systems for hardware-software codesign [Heusinger 93].
- VHDL is syntactically complex. VHDL has a rather irregular, handcrafted syntax, tailored to the conception by designers, not by machines. This is why it is a good benchmark for our specification language.
- VHDL is semantically complex. VHDL inherits from ADA constructs of a modern, structured programming language like modularization, abstract data types, a rich type system, and polymorphism. It adds its own concepts to describe hardware specific data types, concurrency, module generation, and composition hierarchies.
- VHDL has built-in design management features. The language offers a concept of dynamic configurations, which is unusual for a programming language, but very useful during the design and test of electronic circuits on various levels of detail. The designer can first create alternative

implementations of a design entity and decide later which one to use at a certain place in her design.

In this section we will see how our specification language can be used to define those parts of VHDL that are relevant to design information management. It is important to note that we do not attempt to give an exact and complete syntax definition of VHDL. This effort would be rather pointless as complete VHDL grammars exist for parser generators like *yacc*. Our goal is rather to give a minimum specification of VHDL that is nevertheless large enough to digest a complete VHDL file and extract important structural information from it. The specification will accept a language that is larger then VHDL, that is, a file accepted by a language processor generated from our specification need not be a valid VHDL file. We assume that a proper syntactical and semantical analysis of design files will be done by a dedicated design tool and need not be the task of a framework service. This assumption radically simplifies our language processors and makes it feasible for a tool integrator to write processors for languages as complex as VHDL.

The primary design abstraction in VHDL is the design entity. An *entity declaration* defines the interface between a given design entity and the system in which it is used. An *architecture body* specifies the implementation of a design entity in terms of the relationships between its inputs and outputs. There may be multiple *architecture bodies* defined for a given *entity declaration*. The selection of a specific implementation for a design entity is handled by *configuration declarations*.

Concurrent statements are used in architecture bodies to define interconnected blocks and processes that jointly describe the overall behaviour or structure of a design. Concurrent statements come in various forms, notably *block statements* to group other concurrent statements and *process statements* which represent single independent sequential processes. Other concurrent statements exist for commonly occurring forms of processes as well as for representing structural decomposition and regular descriptions.

VHDL provides a rich type system encompassing scalar, composite, access, and file types much like ADA. *Subprograms* may be used to define algorithms. Object types provided are *constants*, *signals*, and *variables*. Objects may be associated with predefined or user-defined *attributes*. Object and subprogram definitions may be grouped into packages, realizing an abstract data type. *Entity*, *configuration* and *package declarations* as well as *architecture* and *package bodies* may be independently analysed and inserted into a *design library*.

### 6.7.2 Syntax specification

Our language specification for VHDL consists of two modules. The first module is concerned with the overall language structure. The granularity is coarse, leaving much detail unresolved. Nevertheless, this module describes enough of the language syntax to recognize the text regions that are to be associated with objects in design information management.

```
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 I USE I entity I architecture I package I config }
} // syntax vhdl
```

A design file contains at least one top-level design unit. In addition, there may be directives to include external libraries and selected objects from these libraries into the scope of the design file. Most of the detail specified in the top-level design units can safely be ignored for design information management. Their names are important, however.

```
rule entity { "entity" Name:ID <\n> guts <\n> END }
rule architecture { "architecture" Name:ID ID EntityName:ID guts END }
rule package { "package" guts END }
rule guts { list { item I ID } }
```

In the rule for entities we have included export directives (the two patterns "<\n>") to show their use. The rule *guts* collectively describes all the internals of design units.

While we completely ignore special characters (remember that the generated scanner is instructed to read over all text for which it has no explicit pattern), certain keywords and identifiers have to be distinguished to find component declarations.

```
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
  | USE
  | component
}
rule component { “component” Name:ID guts END }
```

Although the granularity at which the top-level specification module is written is sufficient to locate design units and extract their names, it is too coarse to analyse configuration declarations. In fact, we need all the details from a configuration declaration, so the specification module for configurations is written at a very fine granularity. Configuration declarations are introduced with the keyword *configuration*, so a rule with this single keyword as right-hand-side provides a suitable anchor for invoking the config specification:

```
rule config -syntax { “configuration” }
```

Structural implementations can declare a component specification and create instances of components. A component thus declared can be thought of as a template for a design entity. The binding of an entity to this template is achieved through a configuration declaration. The declaration can also be used to specify actual generic constants for components and blocks. So the configuration declaration plays a pivotal role in organizing a design description in preparation for simulation or other processing.

The declarative part of a configuration declaration allows the configuration to use items from libraries and packages. The outermost block configuration in the configuration declaration defines the configuration for an architecture of the named entity.

```
syntax config {
 token comment -ignore    -pattern <"--" .* $>
 token IDENTIFIER         -pattern <[_a-zA-Z] [_a-zA-Z0-9]*>

 rule configuration_declaration {
  "configuration" configName:IDENTIFIER
  "of" interfaceName:IDENTIFIER "is"
      list { use_clause } block_configuration
  "end" opt { IDENTIFIER } ";"
 }
} // syntax config
```

Note that a nested specification module is completely self-contained. We have to specify a fresh set of comment conventions, tokens, keywords, and special characters for each module we define.

Within the block configuration for an architecture, the submodules of the architecture may be configured. These submodules include blocks and component instances. A block is configured with a nested block configuration. Where a sub-module is an instance of a component, a component configuration is used to bind an entity to the component instance. Note the extensive use of tags on rule items. We will see shortly how these are used in both import and export specifications.

```
rule block_configuration {
  "for" implName:IDENTIFIER
      list { use_clause }
      items:list { configuration_item }
  "end" "for" ";"
}
rule configuration_item {
  block_configuration
 | component_configuration
}
```

```
rule component_configuration {
   “for” component_specification
        binding:opt { “use” binding_indication “;” }
        block:opt { block_configuration }
   “end” “for” “;”
}
rule component_specification {
   instantiation_list “:” componentName:IDENTIFIER
}
rule instantiation_list {
   repeat -separator “,” { instanceName:IDENTIFIER }
  | “others”
  | “all”
}
rule binding_indication {
   entity_aspect
        opt { “generic” “map” association_list }
        opt { “port” “map” association_list }
}
rule entity_aspect {
   “entity” “work.” childInterfaceName:IDENTIFIER
   opt { “(“ childImplName:IDENTIFIER “)” }
  | “configuration” “work.” childConfigName:IDENTIFIER
  | “open”
}
rule use_clause { “use” “;” }
rule association_list { “(“ “)” }
```

### 6.7.3 Import: Creating objects during parse tree traversal

In Section 6.7.2 we have given the language specification for VHDL configuration declarations. Now we want to use this specification as the basis for parsing a configuration declaration, such as the test bench configuration of Figure 14 on page 76, into the corresponding graph of objects according to the DIM conceptual schema. To achieve this, we write an extension language script that traverses a parse tree automatically created by the VHDL parser module, and creates the DIM objects via extension language commands bound to the DIM PI as soon as enough information is available to fill all attribute values of the created objects. Apart from the parse tree, no data structure is necessary.

We assume that the node identifier of the tree root has been assigned to variable *tree* by the following statement

```
set tree [[vhdl T -file dp32.vhdl] root]

$tree all -type configuration_declaration -var n1 {
        $n1 one -tag configName -type IDENTIFIER {
              set configName $text }
        $n1 one -tag interfaceName -type IDENTIFIER {
              set interfaceName $text }
        $n1 one -type block_configuration n2 {
              doBlockConfig \
                    $n2 $interfaceName $interfaceName $configName
        }
}
```

This top-level loop finds all nodes of type *configuration_declaration*, extracts the configuration name and the name of the associated interface from them, and processes the single block configuration contained in them by calling the procedure *doBlockConfig*. Configuration declarations always contain a single block configuration which configures an architecture body. The other two *one*-statements identify the individual identifiers found in a configuration declaration by checking their tags and exploiting the fact that the text of a matching node can be directly accessed through the text variable from within the action code. No node identifiers are required.

The procedure *doBlockConfig* has the following code:

***Algorithm 2. doBlockConfig***

| | | |
|---|---|---|
| in | n1 | identifier of a block configuration |
| in | path | list of objects that represent the current nesting level |
| in | interfaceName | name of the interface of this block |
| in | configurationName | name of the outermost configuration declaration |
| out | configuration identifier | the identifier of the block configuration created |

```
proc doBlockConfig {n1 path interfaceName configName} {
 set C [Configuration #auto -Name $configName]

 $n1 one -tag implName -type IDENTIFIER { set implName $text }

 $n1 all -type component_configuration -var n2 {
     $n2 one -tag componentName -type IDENTIFIER {
         set componentName $text }
     $n2 one -type entity_aspect -var n3 {
         $n3 one -tag childInterfaceName -type IDENTIFIER {
             set childInterfaceName $text
         }
         set childImplName “”
         catch { $n3 one -tag childImplName -type IDENTIFIER {
                 set childImplName .$text
         }}
     }
```

```
catch[*] { unset CC }
$n2 all -type block_configuration -var n3 {
      set nestedC [doBlockConfig
            $n3 $path.$implName
            $childInterfaceName $configName]
      if {$childImplName != “”} {
            # check whether the block configuration just created
            # is for childImpl
            $n3 one -tag implName -type IDENTIFIER {
                  set nestedImplName $text }
            if {“$childImplName” == “.$nestedImplName”} {
                  # create a component configuration
                  # that points to the block configuration
                  set CC [ComponentConfiguration #auto
                        -Name
                        “cc:$path.$implName.$componentName”
                        -Configuration $C
                        -Component
                        $interfaceName.$implName.$componentName
                        -Child $nestedC]
            }
      }
}
if [catch { set CC }] {
      # no nested block configuration has configured the childImpl
      # create a component configuration
      # that points to the childimpl directly
      set CC [ComponentConfiguration #auto
            -Name “cc:$path.$implName.$componentName”
            -Configuration $C
            -Component
            $interfaceName.$implName.$componentName
            -Child $childInterfaceName$childImplName]
}
```

[*] *Catch* is a Tcl command that evaluates its parameter as usual but “catches” error conditions to avoid that the execution of the whole script is terminated.

```
            # don't search for component_configuration's
            # in nested block_configuration's
            continue
      }
      return $C
    }
```

Here, the parameters for the creation of a component configuration are simplified in that components are referenced by their path name instead of a DIM object identifier retrieved by a query operation. In brief, the purpose of procedure *doBlockConfig* is to resolve a single, possibly nested block configuration statement. It mainly consists of a big loop over all component configurations. The *continue* statement at the end of the loop prevents the traversal from implicitly diving into nested block configurations because these are to be handled explicitly. For each component configuration, a check is made whether a corresponding block configuration exists. If so, this block configuration is recursively resolved and used as child design object for the component configuration. If no such block configuration exists, the named implementation is used as child design object for the component configuration. The net result of letting this code operate on the parse tree created from a configuration in the DP32 processor is the object graph depicted in Figure 14 on page 76.

### 6.7.4 Export: Linking types to parse trees

To export an object graph managed by the DIM service, the type of each object to be exported has to have an *unparse* method that creates a parse tree for objects of this particular type. *Unparse* methods are implemented using the *unparse* functions generated for each parser module. Export simply involves the invocation of the unparse method on a selected DIM object and writing the resulting parse tree into a design file.

For a configuration, an enclosing VHDL *configuration_declaration* has to be created first (we assume the variable *C* to contain the DIM object identifier of the configuration to be exported):

```
[config::unparse -rule configuration_declaration
  -name { -code $C Name } -entityRef { -code [$C Interface] Name }
  -block_configuration { -node $C unparse } ] export $VHDLfile
```

Table 11 shows the unparse methods for the types *Configuration* and *Component-Configuration* in our conceptual schema. Every configuration visited is emitted as a nested block configuration in the VHDL file. Depending on whether the child design object of a component configuration is an implementation or a configuration, the respective VHDL binding indication is emitted.

| DIM type | unparse method |
|---|---|
| Configuration | **return** [config::unparse **-rule** block_configuration |
| | -implName { **-code** [$this Implementation] Name } |
| | -items { **-list set** ComponentConfigurations } ] |
| Component-Configuration | **switch** [$Child **info class**] { |
| | Implementation { |
| | **return** [config::unparse **-rule** component_configuration |
| | -component_specification { **-code** |
| | **concat** "all:" [$Component Name] } |
| | -binding { **-code** |
| | **concat** “use entity” [$Child Name] } ] |
| | } |
| | Configuration { |
| | **return** [config::unparse **-rule** component_configuration |
| | -component_specification { **-code** |
| | **concat** “all:” [$Component Name] } |
| | -binding { **-code** |
| | **concat** “use entity” |
| | [[[$this Implementation] Interface] Name] } |
| | “(” [[$this Implementation] Name] “)” |
| | -block { **-node** $Child unparse } ] |
| | }} |

***Table 11.*** *Implementations of unparse methods for configurations and component configurations. Only the alternatives for implementations and configurations are shown. The unparse method for component configurations that bind to an interface are not shown.*

## 6.8 Related work

In this section we will look into the work done by others related to the topics of language processing in the context of EDA frameworks. First, we review the support for design description languages in EDA frameworks together with the integration level (cf. Section 2.5) on which such a support is given. The JESSI Common Framework (JCF, [Kathöfer 92]) supports only a-priori tool encapsulation based on the copying of design files between a design data repository and the design tools. As no concern is given to the contents of design files, data integration in JCF is on carrier level. EDA Frameworks like Mentor Graphic's Falcon Framework [Mentor] provide format converters between their native design formats and standard or tool specific formats. However, there is no toolkit support for building new converters besides the standard UNIX tools.

The Nelsis CAD Framework [Dimes 93b] supports tool integration on a number of different levels, with a set of interface functions that allow to access design data at varying granularities:

| granularity | interface functions |
|---|---|
| file-based | dmGetPathOfStream, dmMoveFileToStream, dmCloseStream |
| stream-based | dmOpenStream, (standard-i/o facilities), dmCloseStream |
| value-based | dmOpenStream, dmGetDesignData, dmPutDesignData, dmCloseStream |

***Table 12.*** *Integration granularities in the Nelsis CAD Framework*

While Nelsis, too, does not provide a toolkit to write language processors for arbitrary design description languages, it offers an extensible set of *format handlers* for common formats. A format handler is a library function that knows how to read and write design data in a specific format. The functions *dmGetDesignData* and *dmPutDesignData* take the format handler needed to process a particular design description language as a parameter. Format handlers provide a uniform interface to design data from a tool's point of view. Unfortunately, support to write new format handlers is restricted to scanf/printf-like functions. This is clearly insufficient for free-format languages like VHDL.

Regarding the problem of framework support for design description languages to find an answer to the question *"how to map between syntax and objects,"* some solutions are offered in the literature. Many "C++" class libraries have methods to dump an object graph to a file and to read such files back into memory. However, the format of such a dump is fixed, whereas the object-oriented database system OBST follows a more flexible approach. Its *Structurer and Flattener* (STF, [Pergande 93]) is a general tool to map between textual files and objects stored in an OBST database. STF translates a single syntax specification into a pair of programs, a structurer to read an object graph from a file, and a flattener to write an object graph to a file (the generation process is similar to the one depicted in Figure 25). The structurer contains a yacc-generated parser, the flattener is constructed from a set of recursive procedures. While the STF approach resembles ours, it can not be applied for the following reasons:

- STF assumes the OBST database system as back-end of the generated language processors. The mapping between (implicit) parse tree and an OBST object graph is compiled into the language processors.
- STF syntax specifications are restricted in that the syntax for a class must be described with a single syntax rule. While the provided EBNF constructs make up for this to a certain extent, in general it is complicated to map a given language specification to STF.
- STF generated parsers are expected to process every single detail from a file. This does not conform with our approach of only considering the bits in a design description that are important to DIM, ignoring most of the details. To provide more flexibility in choosing the parsing granularity, we provide modular syntax specifications.

*Txl* [Carmichael 92] and *ParsesraP* [Beaty 94] are two other systems that also provide a single specification language for both input and output. Both, however, operate as filters to translate between two different source representations. There is no documented programmatic access to the intermediate parse tree.

Graver describes the system *T-gen* to automatically generate string-to-object translators in the context of the Smalltalk programming environment [Graver 93]. T-gen supports a large class of grammars (LL(1), SLR(1), LALR(1), LR(1)). The generated translators are Smalltalk objects and either create two variants of parse trees, derivation trees or abstract syntax trees from their input. While derivation trees contain all intermediate non-terminals created during parsing, abstract syntax trees are pruned by parse-tree builder directives (PTBs) inserted in the syntax specification by the programmer. PTBs can be used to explicitly construct a parse tree node or to control the flow of information between tree nodes. For example, there is a PTB *liftRightChild* to help flattening the derivation tree. Consider a set of rules of the form*

```
ArgList    : Arg ArgList      { liftRightChild }
           I Arg              { ArgumentListNode };
Arg        : <argument>       { ArgumentNode };
```

The capitalized PTBs *ArgumentNode* and *ArgumentListNode* create tree nodes. The PTB *liftRightChild* prevents that a new tree node is created. Instead, the tree associated with the right-most symbol on the right-hand side is used as tree for the symbol on the left-hand side. The other right-hand side symbols are appended as children to this tree, thus effectively flattening the derivation tree to a list. Our system could benefit from the introduction of PTBs. However, because we skip large parts of an input file, the parse trees generated by our parsers are generally small. By means of our declarative operators *all* and *one* we create the illusion of a pruned tree that only contains relevant information for the programmer. This approach has the advantages that we can keep the syntax specification completely free of manually inserted action code. All "tree pruning" is done from extension language scripts.

Certainly, there are more sophisticated systems to process parse trees, yet all of them insist on processing files down to single lexems, as, after all, this is required for compiler construction. They also have limitations which make them unusable as general toolkits for building language processors to encapsulate design tools. *Txl* [Carmichael 92] relies on its own internal representation of parse trees and provides no documented hooks to extract information from such trees. *Puma* [Grosch 91] uses

*. We use *yacc's* notation here. Curly braces contain action code

Modula2 as implementation language and relies on the use of its associated compiler frontend. A viable alternative to our parse tree operators might have been the use of *Sorcerer* [Parr 94], but we prefer a solution using an interpreted language in which framework administrators or even designers can easily create new or modify existing mappings without the need for recompilation.

# 7. Case Study: Design system for high-level synthesis

## 7.1 Overview

In this chapter we show how to construct a design environment using the encapsulation methodology presented in this thesis. The purpose of this case study is to show that with the new framework service for design tool encapsulation and a reasonable effort for wrapper writing a design environment can be constructed. The encapsulation service has to incorporate the following elements:

- a design information management system implemented on an existing CAD framework
- language processors generated from HDL specifications
- a flexible extension language

We show that such an environment provides added value to the designer when compared to simple a-priori tool encapsulation based on design files. Apart from the fact that such a design system was indeed constructed, we report on the size of the extension language scripts that had to be written to build this environment.

The design environment comprises synthesis and simulation tools for electronic design, starting with a behavioural design description on algorithmic level and resulting in a structural description on logic level. It can be extended with tools to

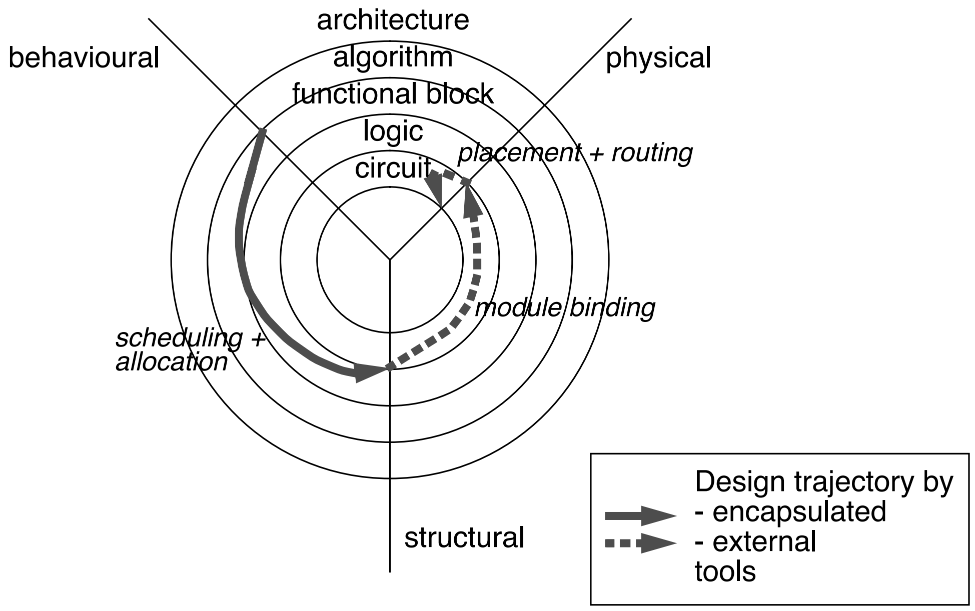


***Figure 28.*** *Design steps possible with the prototype design environment*

transform the logic-level netlist description to a physical layout (Figure 28). Design information management is implemented using the Nelsis CAD Framework. The following design tools have been encapsulated:

- *Import*. Processes a set of VHDL files with behavioural or structural design descriptions and stores them as separate structural and representational information in the framework's design information management database.
- *Text editor.* The design environment supports multiple edit-compile-debug cycles which are typical for the design of complex systems. A text editor can be used to create an initial design description and feed it into the design information management system for administration. With the management of individual design units in the context of a comprehensive web of relationships, though, it seems more advantageous to create and edit design objects directly in the context of this web. Our implementation supports both approaches.
- *Synopsys design compiler.* The design compiler is a design environment in its own right, featuring a graphical user interface, a powerful scripting language and internal, binary design formats. It is able to read and write design descriptions in a multitude of formats for facets of a design like schematics, finite state machine descriptions, behavioural design descriptions, or structural netlists. Its purpose in our design environment is to synthesize a behavioural design description on algorithmic level, written in VHDL by a designer, into a structural design description on logic level, also described using VHDL.
- *Synopsys VHDL analyser.* The VHDL analyser is a batch tool. Its prime purpose in our design environment is to check VHDL files for syntactical errors and thus to guide the designer in constructing VHDL files acceptable to the other design tools and to language processors for design file export. Remember that we assume syntactically valid design files to greatly reduce the size of our language specifications. The VHDL analyser also produces an intermediate file with the analysis results that is needed by the VHDL debugger.
- *Synopsys VHDL debugger*. Much like debuggers for programming languages in software development, the VHDL debugger allows to analyse the runtime behaviour of a compiled design description. The VHDL debugger accepts hierarchical design descriptions on all levels of detail in all domains that are supported in VHDL. It allows to read stimuli, generate reports, single step through a design and examine the contents of variables and signals.

Simulation results are output in a proprietary waveform format that can be visualized by a separate tool.

- *Export*. Export takes a (hierarchical) design from the database and exports it to the file system as a set of VHDL files. This is needed because the design of an electronic system is not completed with the creation of a logic-level structural netlist. Further processing is necessary to transform this netlist into a physical mask layout with the aid of associated technology libraries. The physical mask layout can then be used as the basis for chip fabrication.

By using the Nelsis CAD Framework as the implementation platform for design information management, the designer can use the following framework tools provided by the Nelsis CAD framework as an addition:

- Graphical database query interface
- Version browser
- Equivalence browser
- Design flow browser

Note that using these framework tools only makes sense for a designer because it was possible to map DIM concepts naturally to Nelsis schema types (Figure 22 on page 93).

## 7.2 Design flow

The design environment arranges the encapsulated design tools in a design flow in which tools are linked to each other via typed data channels (Figures 30, 30). The overall structure reminds of a bus-based system in electronic designs in which design objects described in VHDL are transferred on the central channel or „data bus“. Design objects flowing between tools on this bus are managed by the design information management system. In addition, most of the tools can import or export design data external to the system. Some tools produce data in an opaque, binary format that, although being associated with a VHDL design object, has no internal structure that is known to the DIM service. The following subsections give a brief overview of the design tasks that are performed in a typical design situation using this design environment.

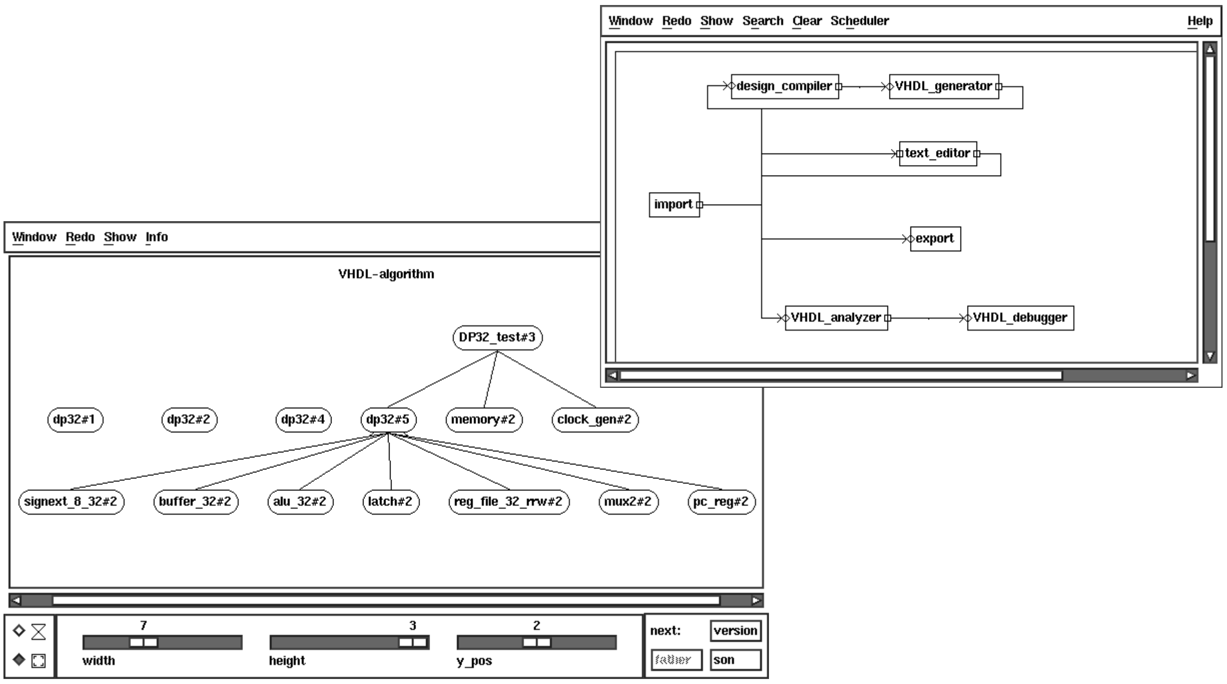


***Figure 29.*** *Screen shot of a typical design situation showing the user interfaces of the two framework tools* hierarchy browser *and* flow browser *working on the DP32 processor.*

## Creating and importing VHDL designs

A behavioural VHDL description on a low level of detail (typically algorithmic or functional block levels) of the design is created either manually using a text editor or a graphical specification tool, or as the output of some synthesis tool operating on even lower levels of detail. The design may be arranged in a single or in multiple design files and typically also references some external library elements. The VHDL files are imported into the DIM service, creating design objects for each entity declaration, architecture body, and configuration declaration, and establishing hierarchy relationships between them. The object graph thus created can be inspected with the hierarchy and version browsers provided by the underlying framework. Individual design objects can be viewed or edited by an encapsulated text editor which is a significant added value to the stand-alone tool suite.

### Analysing design objects

The Synopsys VHDL debugger *vhdldbx* can be invoked on the top-level design object to validate the design. The debugger does not read VHDL objects directly but rather expects analysed **.sim* files produced by the VHDL analyser *gvan*. This design step reveals any syntax errors that remain in the design objects. If an error is encountered *gvan* pops up a window and allows to edit the design object.

### Simulating design objects

After *gvan* has analysed the VHDL objects, *vhdldbx* can be invoked on the top-level or any intermediate-level design object. The design object hierarchy is traversed for this and exported as proper VHDL files into a temporary directory. Each VHDL file is accompanied by the intermediate file created by *gvan*. Within *vhdldbx* arbitrary simulation steps can be performed to validate the design.

### Synthesizing design objects

Once the design is found to be correct it can be synthesized. Synopsys' design analyser is invoked for this task and fed with a set of VHDL files that have been extracted from the object hierarchy to be synthesized.

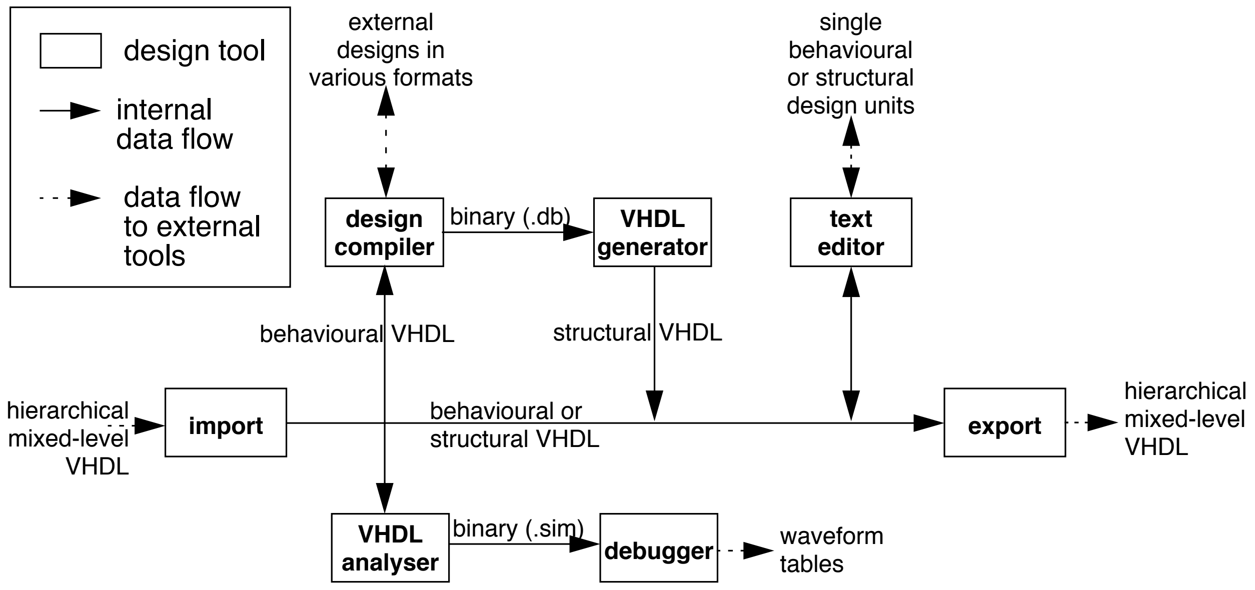

design tool
internal data flow
data flow to external tools
external designs in various formats
single behavioural or structural design units
design compiler
binary (.db)
VHDL generator
text editor
behavioural VHDL
structural VHDL
hierarchical mixed-level VHDL
import
behavioural or structural VHDL
export
hierarchical mixed-level VHDL
VHDL analyser
binary (.sim)
debugger
waveform tables

***Figure 30.*** *Design flow. Dashed lines denote data channels which either enter or leave the environment and provide links for external processing.*

### Deriving new alternatives

The synthesized, structural design is at first stored in intermediate **.db* files. This allows to perform an arbitrary number of synthesize steps until the result is satisfactory. An explicit design step *derives* a new VHDL description from these **.db* files by invoking the batch version of the design analyser *dc_shell* with an appropriate script in its native scripting language. Now the synthesized design objects may be inspected using either the text editor or the debugger.

### Export

The final step in a typical design flow is to export the synthesized design into external VHDL files for further processing by external tools.

## 7.3 Building the environment

Building a design environment requires a number of tasks which have to be considered carefully. In the following, we will describe some of the decisions that lead us to constructing the design environment in the way we have.

### 7.3.1 Selection of the design tools to be integrated

The selection of the design tools to be integrated is probably the most important step because it determines to a large extent what the final design environment can be used for. It is not only important to select tools that offer a suitable functionality for a specific design task. The tools must also support a given design methodology. It is useful, for example, to already be able to simulate a behavioural description of a design and compare the result of this simulation with the results of a simulation conducted on some structural version of the same design. This implies that either a single simulator/debugger is available that can simulate both kinds of representation, or that two simulators are used which produce comparable results. We have chosen to base our design environment on a commercial suite of tools that contains a multi-level simulator which is capable to simulate both behavioural and structural VHDL files and to produce comparable results.

### 7.3.2 Dimensions of integration

A vital property of the selected tool suite is that the tools are *data-integrated*, that means that the output data of one tool can either be directly fed into another tool or that appropriate converters are available to convert the output of the first tool to a data format understood by the second tool. Encapsulated tools are *not* automatically data-integrated. In principle, our framework service with its toolkit for building parsers and unparsers can be used to construct format converters that read a design description in one description language, convert it into some language independent format and emit a description in another language. On the other hand, the granularity at which we manage design objects is too coarse to serve as intermediate, language independent design representation. This is of course exactly what we intended to keep language specifications small. The Synopsys design tools all support VHDL as a common data format. Some tools do not directly read or produce VHDL but need a pre- or post-processor to interface with VHDL. Further formats can be used by the tools but are not supported for integration purposes in our design environment. To summarize, we do not attempt to build any format converters in addition to the ones supplied with the tool suite.

Another desired property of a tool suite is that it is control-integrated. Control-integration means that tools can exchange messages and react to them in an asynchronous manner. Exchanging messages can be used in a design system in a multitude of ways. Some applications are:

- controlling a design tool after it has started, sending it additional commands to execute tasks not anticipated at tool start-up time
- logging execution progress to a design monitor tool to record design history, thus achieving tracing encapsulation (Section 2.8 on page 30).
- linking two tools, e.g. a batch simulator and a graphical design browser, so that the view displayed by the graphical browser always focuses on the currently simulated design objects
- delegating design tasks between tools which allows to structure the design environment into modules

The language processing facility provided by our encapsulation service does not aid in control-integration. Most tool suites and frameworks today, e.g. the Synopsys graphical environment or the Nelsis framework, provide some kind of messaging system based on proprietary message formats. However, standardization is well under way, either as messaging system specific to a particular operating system

(DDE for Microsoft Windows, ToolTalk for SunOS, Softbench for HP), or as an industry standard (CFI ITC, OSF DCE). With the current state of technology, we have to rely on the tools already being able to talk to each other. With regard to control-integration this leads to a situation in which there are tightly integrated tool clusters with well-established inter-tool communication links and more loosely coupled links in between. We also encounter this situation in our design environment. Nelsis has a system of *framework messages* that is used internally to keep the views displayed by the graphical browsers up-to-date with respect to each other and the framework database. Some of the Synopsys tools (debugger, hierarchy browser, waveform display) are integrated with a proprietary messaging system.

As a further option a common look and feel for the design tools and framework tools may be desirable, achieving the lowest level of presentation integration. The most currently achievable here is the use of a Motif-compliant user-interface style for the tools with graphical user interfaces. Both Nelsis' and Synopsys' graphical tools have a Motif-compliant user interface (cf. Figure 30).

### 7.3.3 Encapsulation tasks

In this section we describe the actual procedures that are used to encapsulate the design tools proper. Two of these procedures (import and export) function as basic services used by all the other procedures. The other procedures serve as wrappers for individual design tools and are more specific to their input and output needs. These procedures are comparable to the tool wrappers used in conventional, file-based tool encapsulation.

#### Importing VHDL files into the design information management service

This procedure takes a set of design files in VHDL, extracts the contained design units as text chunks and associates these text chunks with design objects in the design information management database. First, each VHDL file is parsed and a parse tree is constructed for it. The parser is generated from a VHDL language specification as described in Section 6.4.2 on page 112. For each design unit (entity declaration, architecture body, configuration declaration) found, the text of this design unit is replaced by a marker in the corresponding design file as described in Section 5.2.5 on page 78. The remaining file trunk is stored as lexical context for the design objects created for each of these design units.

Then, the parse trees are traversed and the following actions are performed:

- For each entity declaration visited, an interface design object is created. The VHDL text of the entity declaration is written as *interface* stream into this design object.
- For each architecture body visited, an implementation design object is created and inserted into the implementation set of the corresponding interface design object. The VHDL text of the architecture body is written as *implementation* stream into this design object. A default configuration design object is also created and associated with the new implementation. Starting from the architecture body, the parse tree is further searched for component declarations. Each component declaration found is inserted as component into the default configuration.
- Each configuration declaration visited is analysed as described in section Section 6.7.3 on page 127. The result is a new configuration design object for each block configuration found in the configuration declaration. Remember that in the case of configuration design objects the original VHDL is not stored with the object. Rather, all relevant information is resolved into attribute values and relationships in the DIM database.

### Export design hierarchies to VHDL files

To export a design hierarchy from the DIM database to VHDL files, the design hierarchy has to be traversed and associated VHDL text has to be emitted. On each level of the hierarchy, the following chain of objects has to be visited:

$$\begin{tikzcd}
\text{interface} \arrow[d] \\
\text{implementation} \arrow[d] \\
\text{configuration} \arrow[d] \\
\text{component} \arrow[d] \\
\text{component configuration}
\end{tikzcd}$$

The component configuration references a design object of either subtype. Interface and implementation design objects are directly associated with their VHDL text chunks. Such a text chunk is inserted into the associated context text and placed into the exported VHDL file. Configurations are unparsed as described in Section 6.7.4 on page 131 and exported. The whole process of exporting a design hierarchy is controlled by a recursive procedure that takes as parameters

- the object references of the root design object *rootId* in the hierarchy and the reference of the currently processed design object *currentId,*
- a regular expression *binaryPattern* for the names of binary streams that have to be exported alongside the actual VHDL design descriptions,
- Tcl code *hierarchyAction* to be executed for every design object in the hierarchy, and
- Tcl code *rootAction* to be executed for the root design object only.

On each hierarchy level, for each interface visited, the interface text is exported. Then an implementation with version status *actual*, or, if none exists, one with version status *working* is visited. For each implementation visited, the interface and implementation texts are exported. If the interface design object contains binary streams whose names match *binaryPattern* these streams are exported as individual files. Then the associated default configuration is visited. For each configuration visited, the interface and implementation texts are exported and the configuration is unparsed and exported.

After all design objects have been exported on a certain level of hierarchy, the recursive export procedure calls itself for each child design object in the current (default) configuration to recursively export all the children in the design hierarchy. Following this recursive descent, on each level of the hierarchy, *hierarchyAction* is executed. If this action results in the creation of new binary files, they are saved as streams in the current implementation design object. If the current design object is the root of the exported hierarchy, *rootAction* is executed after that. Binary files that have not yet been collected by a *hierarchyAction* are collected now and associated with the root object in the exported hierarchy.

The remaining encapsulation tasks operate in the structure imposed by the recursive export procedure and add appropriate code fragments for *hierarchyAction* and *rootAction*. These code fragments may also call the import procedure to import VHDL files resulting from design tool runs in either of these actions.

## Analysing VHDL files

To analyse design objects the Synopsys tool *gvan* is invoked in a *hierarchyAction* code fragment. When a syntax error is encountered, *gvan* emits the file name and line number of the error. A text editor is started in which the designer may correct the error. Syntax check and editing is repeated until *gvan* reports no more errors. The corrected VHDL design files are then imported into the DIM database by calling the import procedure. Along with them the **.sim* files generated by *gvan* are saved. These files are needed as input files for the debugger. The import either overrides an existing *working* version or creates a new version of the corrected design object.

## Synthesizing structural from behavioural descriptions

Synthesis is conducted by invoking the Synopsys *design_analyser* in a *rootAction* code fragment. Some cooperation is required from the designer as a lot of actions can be performed using the graphical user interface of this tool, resulting in output files not noted by the encapsulation scripts. After successful synthesis, the designer is requested to write out the synthesis result as a binary **.db* file. The export procedure collects these binary files and associates them with the hierarchy root. A separate VHDL generation tool is invoked explicitly by the designer to convert such **.db* files to VHDL text that can be imported into the DIM component. This has the advantage that some iterative synthesis steps can be performed on the intermediate files which are read much faster by the *design_analyser* than are VHDL files. The VHDL generator actually is a script in *design_analyser's* script language, fed into the text-only interface of the tool. The resulting VHDL files are imported into the database as usual.

## Debugging a design

Debugging a design is performed by running the Synopsys tool *vhdldbx* in a *rootAction* code fragment. The debugger needs the exported VHDL files only to show currently debugged VHDL text in its text window. The actual design data are read from *.sim files resulting from previous VHDL analysis. Any binary files produced by the debugger are saved along with the hierarchy root but are otherwise left uninterpreted. They can be exported by standard framework means and fed to waveform displays, result comparators or other post processing tools.

| | Module | | Lines of code | | |
|---|---|---|---|---|---|
| | | | Language spec. | Tcl | C++ |
| **1** | Parse tree implementation | | | | 441 |
| | DIM PI on top of Nelsis | | | | 971 |
| | Schemas for demand loading | | | 94 | |
| | | Parse tree | | 32 | |
| | | DIM | | 62 | |
| **2** | VHDL language specification | | 123 | | |
| | VHDL file import | | | 150 | |
| | VHDL file export | | | 127 | |
| **3** | Execution protocols | | | 211 | |
| | | Analyse (calls *gvan* and *vhdldbx*) | | 36 | |
| | | Compile (calls *design_analyser*) | | 37 | |
| | | Derive (calls *dc_shell*) | | 39 | |
| | | Edit (calls text *editor*) | | 99 | |

***Table 13.*** *Lines of code in the prototype implementation of the encapsulation service*

## 7.4 The implementation

The actual implementation of the prototype design environment is structured according to Figure 10 on page 56. It is built around a central Tcl interpreter that provides bindings to the underlying design information manager and to the various modules of the encapsulation service implemented in “C++”. The flow of control and design data between design information manager and encapsulated tools is conducted by Tcl scripts. Table 13 shows the size of the prototype in lines of code. The size of the individual modules is generally small. Modules under (1) are core modules of the encapsulation service and are invariant to supported design description language or encapsulated design tools. Modules under (2) are invariant to design tools but have to be written for each new design description language supported. Only modules under (3) depend on the actual design tools. These modules contain code that is invoked from within the *hierarchyAction* and *rootAction* code blocks of the *export* module. They invoke procedures from the *import* module to import result files in to design information management. To summarize, the amount of code necessary to encapsulate design tools with the aid of our new encapsulation service is similar to what is needed with classical shell wrapper encapsulation. The effort needed to add support for a new design description language is also low.

## 7.5 Discussion

First, we check to what extent our implementation satisfies the requirements stated in Section 1.6. Requirements **R1** to **R5** are well covered by our conceptual schema. Requirement **R6** is covered by the application of the syntax specification language and associated language processor construction toolkit. It must be emphasized that we do not attempt to improve tool interoperability by writing converters to translate between different design representations. Our main technical goal is to improve interoperability between tools and framework components. This improves transparency between encapsulated and integrated tools in the resulting design system. Finally, by selecting a widely used, leading edge commercial synthesis suite **R7** could in fact be satisfied. We let the analysis of our implementation be guided by the categories of Chapter 2, starting with *granularity*. The following table summarizes the different granularities in our implementation:

The separation of object-based storage and file-based transport of design data between DIM framework service and design tools is a major achievement of our approach. Storage with at least object granularity is an important prerequisite to being able to do design information management at a granularity that is natural to designers and effective to be used when a web of relationships between design objects is to be built and maintained. As mentioned above, this raise in granularity only improves interoperability between framework and tools and does not in itself provide better tool interoperability. In this area we rely on standard design description languages like VHDL. Using such a standard language also ensures tool interchangeability at least as far as design data exchange is concerned.

Unfortunately, our approach contributes little in the area of control integration. In this area tools should provide programmatic access to their inner workings. Although the Synopsys tools do extensive logging of design actions and are busily engaged in inter-process communication, little of this can be used to interfere with a tool once it is started. We have experimented with ways to trace information written into the log files, but with standard i/o buffering one can never be sure if logging information appears synchronously with design processing steps. Also trying to control a running tool by sending it commands during runtime was a frustrating experience. Design analyser, for example, comes with an extensive programming language with a rich set of configuration options, control flow statements, and design commands. Unfortunately, the variant of design analyser with graphical user interface does not provide a channel to feed it commands at run time by another program. To conclude, a minimum of programmable observability and controllability is required in a design tool to achieve any kind of control integration beyond simply starting and stopping of tools. Users find completely new metaphors of employing tools once they are able to attach external programs to them (e.g. [Silva 93]) so even the vendors profit through greater sales. Publishing tools like e.g. FrameMaker (through ONC/RPC) or many MS Windows programs (through DDE and OLE) already provide this kind of openness.

Presentation integration also suffers through this lack of control interoperability. While the Synopsys tools as well as the Nelsis Framework tools are able to update window content amongst themselves with new events arriving, the two tool groups are isolated from each other in this respect.

| Mechanism | Granularity |
|---|---|
| storage | object-based |
| transport | file-based |
| processing in the DIM service | object-based |
| processing in the tools | value-based |
| browsing | object-based |

***Table 14.*** *Granularities applied to different mechanisms*

Finally, process integration could be much finer if only the programming languages offered by the design tools could be used by external programs or wrappers to control the design tools. Currently, only complete sub-functions of the tools as configurable by command line options or initialization scripts can be used as building blocks to construct a design process.

Still, we feel that raising the granularity of data storage in a framework's DIM service is a major step to improving end users' productivity and will help to foster the acceptance of open frameworks with off-the-shelf tools as opposed to single-vendor design systems.

The following table summarises our achievements in the integration categories as introduced in Chapter 2:

| Category | | Achievement |
|---|---|---|
| Inter-operability | framework-tool | Raised granularity |
| | tool-tool | Not directly supported. Must pre-exist |
| Inter-changeability | | Not directly supported |
| Dimensions | data | Major asset |
| | control | Better infrastructure through Tcl extension language. Not exploited in the prototype |
| | presentation | Not directly supported |
| | process | Conceptually supported by execution protocols. Not exploited in the prototype due to lack of observability/controllability in the tools |

| Category | | Achievement |
|---|---|---|
| Levels | carrier | Original level for file-based encapsulation |
| | lexical | Lexical contexts |
| | syntactical | Syntactical contexts |
| | semantical | Conceptual model for structural information |
| | method | Not addressed |
| Granularity | | Raised (cf. Table 14) |
| Observability | | Conceptually addressed (cf. "tracing encapsulation", page 31). Not exploited in the prototype |
| Controllability | | Not addressed |

## 7.6 Related work

To show that this is indeed the case, we look at two other efforts to build integrated design environments. The first example is the Synopsys tool suite [Synopsys] itself, with its extensive range of tools for design synthesis and simulation. Several standard design description languages interface the tool suite to the outside world, amongst them VHDL, Verilog, and EDIF. We will only look at VHDL here. In the Synopsys tools suite there are two major tool clusters, namely the design compiler for synthesis and the simulators and graphical browsing environment with symbol and schematic editors for browsing and simulation. Data exchange between the two clusters is via two alternative paths:

- Opaque **.db* files created by the design compiler are used to transport graphical information such as generated schematics from the design compiler to the browsing and simulation subsystem. These files have to be converted by a tool *db2sge* into the native format of the browsing subsystem.
- Behaviour and structural design information are exchanged between the two clusters via VHDL files. While the design compiler directly reads VHDL files, simulator and debugger rely on intermediate **.sim* files produced by an analysis tool.

There is no control integration between the two tool clusters. Within the browsing and simulation environment, a lot of control messages are exchanged between schematic editor, waveform displayer, and debugger to maintain the illusion of different views on single design objects. The message types and format is unpublished and so can not be exploited for framework integration.

With respect to design information management, the design compiler largely ignores configurations in its VHDL input but applies simplified, default bindings by name. It can, however, generate proper VHDL configuration statements as output which can be read by the debugger and simulator. By supporting configurations already in the DIM component and resolving them on export, we can make these differences in configuration support in different tools largely transparent to the designer.

Another project which is interesting to compare to our approach is the effort to encapsulate some Cadence tools into the JESSI Common Framework (JCF, [JCF 94b]). The encapsulation relies entirely on file-based storage and transport between the JCF and encapsulated Cadence tools. While JCF only provides file-based granularity, Cadence tools are integrated into the Cadence Opus framework at object- to value-granularities. The encapsulation relies on the standard JCF approach (outlined in Section 2.8) to export a set of predefined files from a framework-managed cell's pool of design descriptions, to invoke some shell wrapper and have it pass control to the design tool, and finally to collect the results. Once control has been passed to the design tool, no communication happens between the tool and the framework until the tool is finished. All design data management on the Cadence side is volatile and effects only the current tool run. The resulting prototype design environment proved unusable for all practical purposes.

# 8. Conclusions

## 8.1 Statement of problem

The foremost concern of designers in electronics in picking a design environment has always been to combine state-of-the-art design tools. Today design tools support designers during all phases of the design, starting with the capturing of a high-level, behavioural design description, through the various transformation and validation stages, finally yielding a low-level description of what has to be physically fabricated. Before the availability of standard and powerful design description languages for all domains and all levels of detail, achieving data integration between tools was a major difficulty. Since their standardization in 1987, the languages VHDL and EDIF have become the Esperanto of electronic design. Today all major EDA tool vendors support one or both languages to capture and exchange design data from the behavioural, structural, and physical domains.

With ever-growing sizes of designs and design teams, design management has moved into focus. EDA frameworks have been introduced to provide data, control, and presentation integration services to help manage large designs. While the first EDA frameworks have focused on data integration, it was realized that for all practical purposes, data exchange through design files solves designers' needs. Since then, the attention of framework vendors has moved to design management. Versatile graphical browsers have been built on top of the basic data management services provided by the frameworks that allow designers to query and manipulate the design state at the granularity of design objects. Design processes can be automated through net-based flow editors. Framework tools communicate with each other through flexible message links.

Today, two facts prevent the effective incorporation of file-based design tools into EDA frameworks:

1. Framework vendors, in an attempt to stay domain independent, have focused on enabling technology for design data management, while neglecting powerful design information management. Especially the VHDL language offers a flexible configuration scheme that has to be mapped to a framework's conceptual schema for design information management.
2. File-based design data exchange has always been considered a bad thing, something that must eventually be overcome. Hence, framework-oriented projects like the German DASSY project and standardization bodies like CFI

have long focused on developing procedural interfaces. Procedural interfaces where thought to rather sooner than later replace design files because of their obvious advantages. Unfortunately, design description languages flourish and are well accepted by designers as well as tool vendors. Languages like VHDL and EDIF have become the prime interface for off-the-shelf tools. Procedural interfaces today are almost always clumsy add-on's, used as technology demonstrators but hardly usable to exchange real designs.

## 8.2 Solution approach

To overcome these problems, we contribute three major assets to the design automation community:

1. Up to today, conceptual schemas for EDA frameworks have been developed separately from schemas that capture the semantics of design descriptions. In fact, there are two nearly disjoint communities working on overlapping topics. We have developed a conceptual schema that explicitly takes into account both, object-based design information management that supports team-oriented design in an integrated environment and requirements that stem from the design description language VHDL. The resulting conceptual schema could be mapped naturally onto the schema of the Nelsis CAD framework. This effort shows that little is in fact needed to conceptually unify framework-based design information management and the use of high-level design languages.
2. Language processing is traditionally regarded as being in the realm of tool vendors who can rely on standard compiler technology to construct design file input and output processors. We advocate the use of language processors not only for detailed design file analysis but also to extract features with the purpose of design information management. Design files need not be analysed down to every single detail to access useful information to be managed by a framework. Instead, we have defined and implemented a specification language and associated toolkit that facilitates the specification and extraction of useful information from standard design files. The toolkit utilizes our specification language to specify both analysis and reconstruction of design files. Our approach is novel both from the compiler construction and design encapsulation viewpoints.

3. While commercial frameworks offer extension languages (e.g. Cadence's *Skill*, Mentor's *Ample*, JCF's *Elk*), design tool *encapsulation* relies on standard UNIX shell wrappers around black-box integrated tools. We have defined the architecture of a new framework service that is based around the publicly available extension language *Tcl* that combines familiar shell-like syntax with easy extendability and close integration with the language processing toolkit. This new service fits nicely into CFI's proposed framework architecture reference model and into the existing Nelsis framework implementation.

The ignorance of the success of design description languages has lead to the strange situation that on the one hand there is mature framework technology, offering powerful design data management and versatile and user-friendly graphical browsers whereas, on the other hand, there are state-of-the-art file-based design tools that do not match. The unfortunate result is that *open* frameworks have started to become one of the unfulfilled promises of the past, similar to the general problem solvers in artificial intelligence research.

## 8.3 Technology developed

In this thesis, we present technology that tries to bring together the opponents. Incorporating the extended conceptual schema and new framework service for design tool encapsulation certainly does not provide the optimal solution to design tool integration, but it provides a missing link that provides wider acceptance of open EDA frameworks amongst designers and tool vendors. In Chapter 2, we have shown that there are more aspects to tool integration than simply being black- or white-box. The conclusion was that fine-grained design information management is not necessary and in fact not desirable for file-based design tools. On the other hand, file-based storage of design data prevents effective design information management. Hence, we propose to store design data at the granularity of design objects, but to import and export them in design files. In Chapter 4 we have described a new framework service that encompasses services for scripting tool wrappers using a language more suitable than standard Unix shells. The most important contribution of this new service is a language independent facility to process design files. While such a facility maintains the domain independence of the framework, it allows to extract design information from design files on import and to recombine text chunks associated to design objects managed by the framework on export. Thus, a framework is free to store design data at the granularity of design objects. At this granularity a web of relationships between design objects may be established and maintained automatically by the framework. Graphical browsers can be used by designers to help understand their design.

This web of objects and relationships is based on the conceptual schema developed in Chapter 5. It has to be emphasized that this schema is based on a formally defined modelling technique. We have chosen to use the Xplain data model over its competitor EXPRESS because of its clear graphical notation and built-in semantic constraints. The conceptual schema is based on the schema of the Nelsis CAD framework and adds just enough types and relationships to support dynamic binding of components to design objects and configurations as offered by VHDL. With this schema it is possible to directly map VHDL design descriptions to a web of design objects.

With architecture and conceptual schema in place, the toolkit presented in Chapter 6 greatly simplifies the construction of language processors for design tool encapsulation. Based on standard compiler construction technology, by exploiting the virtual memory management facilities of modern operating systems, the parsers generated by this toolkit can skip large regions of input text and nevertheless associate this text with design information extracted from a design file. Modular grammars are introduced as a handy means to choose different granularities of analysis for different regions of the input text, further reducing the necessary amount of syntax detail to be specified. The generated parsers automatically create parse trees that can be traversed from Tcl scripts to gather design information necessary to create design objects in the design information management component of the framework. We have renounced the introduction of complicated tree pruning operators because due to the little detail extracted from the input text, parse trees tend to be small. Rather, the two dedicated extension language methods on parse tree nodes *all* and *one* serve as filters during parse tree traversal to help look only at important information.

The reverse direction, exporting design objects to valid design description files, is supported by two mechanisms. Lexical contexts are stored and managed in the framework and are used as file trunks in which text chunks associated with design objects are inserted. In addition, object types in the design information management component can be associated with tree constructors automatically generated from the same syntax specification as the parsers. Such a constructor knows which attribute values to convert into syntax when traversing an object graph to export a design description file. We used this toolkit to generate a parser and tree constructors for VHDL. The syntax specification for VHDL has well below 150 lines, needing a few hundred lines of Tcl code to link the parse trees to the conceptual schema. This amount of coding is well in the range of work accepted by designers as necessary to create a seamless design environment in which it is possible to conduct productive design work.

Chapter 7 describes the implementation of design environments based on our conceptual schema, the encapsulation service with VHDL language processors generated by the toolkit, some commercial design tools from Synopsys, and alternatively, the object-oriented database management system ObjectStore or the Nelsis CAD framework. The implementation based on Nelsis proved most useful because it allows to use the powerful Nelsis browsers and design tools already integrated with Nelsis together with VHDL-based, commercial synthesis and simulation tools. A first prototype of this implementation was demonstrated at the '94 JCF Framework Workshop in Karlsruhe and was well received [Schettler 94a]. The concepts developed in this thesis were also presented at the International Conference on CAD [Schettler 94b].

## 8.4 Advantages

The technology presented in this thesis provides better data integration between encapsulated design tools and a design information management component of an EDA framework. Design tool encapsulation based on design files may become obsolete one day, when programming interfaces are mature and flexible enough to cover all aspects of an electronic design description. With the current euphoria about the integrative nature of standard design description languages like VHDL (cf. for example [Hüwel 92], [Synopsys 92], [Meersman 94]) this may still take some years. Meanwhile, the conceptual schema we have developed can serve as a starting point for framework developers and standardization bodies like CFI to extend their framework schemas with concepts that are especially important when trying to support VHDL. Amongst these are separation of interface and contents, distinction of domains and levels of detail and the support for the configuration of whole occurrence trees. On the other hand, language standardization may well take into account aspects that are relevant for integrated design environments. While VHDL has gone a long way in this direction by separating interfaces and implementations and the availability of hierarchical configurations, finer control would be desirable over the selection of implementations based on status or version information.

## 8.5 Disadvantages

Our approach currently lacks an effective means to establish equivalence relationships between input and output data of a design tool. The wrappers could try to deduce some of these relationships by matching names in the result files with objects already in the database [Blackburn 85]. This is a rather clumsy approach because it tries to deduce information from the design data that the design tools already have. Here in fact some help from the encapsulated tool would be called for. A comparatively non-intrusive approach would be to have the tool register its object accesses by a programming interface similar to the one used by Cassotto in his VOV system [Casotto 90]. But this would deviate from pure design tool *encapsulation*. A different approach was suggested in Section 2.8, namely to let the framework emulate a network file system on which the tool would operate.

As discussed in Section 7.5, our approach contributes little to improving control integration in general. Control integration has long been hampered by the lack of sufficiently high-level standards for inter-process communication. As such, tool vendors had to develop proprietary means to enable inter-tool communication (ITC). Unfortunately, tool vendors also seem to consider a good but proprietary ITC mechanism as a major selling point for their tool suites. The interesting work of Silva et al. ([Silva 93]) suggests that opening up these interfaces may be a good selling point for tool suites because users then are able to come up with completely new applications of tools that the vendors have not originally thought of. This unfavourable situation may change with the introduction of vendor technology like ToolTalk [ToolTalk] or industry standards like CORBA [OMG 90], provided that tool vendors accept the need for open, programmable interfaces to their tools.

## 8.6 Outlook

We have reached our goal of bridging the gap between file-based design data interfacing of design tools and object-based design data management in EDA frameworks. With the approach presented, even file-based design tools can benefit from the sophisticated browsers offered by modern framework technology. Still, the system we have presented in this thesis can by no means represent the final word in tool encapsulation. While a suitable next step on the framework side would be the provision of a network file system on top of a design information management system, further progress largely depends on the improvement of observability and controllability of design tools. The Synopsys tools suite we have selected as base for our design system leaves much room for improvement in this respect. The incorporation of a commercial tool suite certainly is a good selling point for our approach. However, the availability of tool source code would have enabled us to cover control integration more deeply than we have been able to do now. Hopefully, users' pressure will lead tool vendors to open up some of their interfaces to the design automation community. Both designers and vendors will profit from this.

## References

[Adams 79]
Adams, Douglas, *"The Hitch Hiker's Guide to the Galaxy"*, Pan Books, London, 1979, p. 135

[Ashenden 90]
Ashenden, Peter J., *"The VHDL Cookbook"*, First Edition, Dept. Computer Science, University of Adelaide, South Australia, July 1990

[Batory 85]
Batory,D.S. Kim,W., *"Modeling Concepts for VLSI CAD Objects"*, ACM Transactions on Database Systems, vol. 10, #3, 1985, pp. 322-346

[Beaty 94]
Beaty, Steven J., *"ParsesraP: Using one grammar to specify both input and output"*, Cray Computer Corporation, 1110 Bayfield Drive, Colorado Springs, Colorado 80906 URL=http://www.craycos.com/~beaty/ParsesraP/ParsesraP.html

[Biliris 89]
Biliris, A., *"Database support for evolving design objects"*, 26th ACM/IEEE Design Automation Conference, 1989

[Blackburn 85]
Blackburn, Robert L.; Thomas. Donald E., *"Linking the Behavioral and Structural Domains of Representation in a Synthesis System"*, 22nd Design Automation Conference, 1985, pp. 374-380

[Brannon 94]
Brannon, Terrence Monroe, *"Survey of Quick Interpreted Languages"*, P.O. Box 5027, Bethlehem, PA 18015, tb06@122e.eecs.lehigh.edu, March 1994

[Bredenfeld 90]
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, USA, 1990

[Brielmann 92]
Brielmann, Maria; Kupitz, Elisabeth, *"Representing the Hardware Design Process by a Common Data Schema"*, IEEE, 1992

[Brown 92]
Brown,Alan W.; McDermid, John A, *"Learning from IPSE's Mistakes"*, IEEE Software, March 1992, pp. 23-27

[Camposano 90]
Camposano, Raul, *"From Behavior to Structure: High-Level Synthesis"*, IEEE Design & Test of Computers, 1990, pp. 8-19

[Carlson 92]
Carlson, Steve; Lehbrink, Wolfgang; Rothenaicher, Peter, *"HDL-orientierte Synthese-Methodik (Teil III)"* (in German), 1992, pp. 82-93

[Carmichael 92]
Carmichael, Ian, H.; Cordy, James R., *"TXL - Tree Transformation Language; Syntax and Informal Semantics (Version 6.0)"*, Kingston, Canada K7L 3N6 - cordy@quis.queensu.ca, 1992

[Carter 91]
Carter, Donald E.; Stilwell Baker, Barbara, *"Concurrent Engineering: The Production Development Environment for the 1990s"*, Addison-Wesley Publishing Corporation, 1991

[Casotto 90]
Casotto, Andrea; Newton, Richard A.; Sangiovanni-Vincentelli, Alberto, *"Design Management based on Traces"*, 27th Design Automation Conference, 1990

[CFI-EL 91]
CAD Framework Initiative, Extension Language Working Group; Timothy Barnes (ed.), *"Extension Language: Core Language Selection"*, Version 0.6, 1991

[CFI-DRPI 93]
CAD Framework Initiative, Design Representation Technical Subcommittee, *"Electrical Connectivity Programming Interface"*, Version 1.0, 1993

[CFI-EII 95]
Teets, John, *"EDA Industry Standards for Design and Test Roadmap"*, in preparation of the EDA Industry Roadmap Workshop Review, CFI EII, TLM, DMM working groups, August 1995
URL=ftp://cfi.org/public/Cfi/Development/Roadmap/EII/roadmap-eii-report.ps

[CFI-FAR 93]
CFI Architecture Working Group, *"Framework Architecture Reference"*, Version 1.2, #91, 1993

[CFI-TES 93]
CAD Framework Initiative, Design Methodology Management Subcommittee, *"Tool Encapsulation Specification"*, Version 1.0, 1993

[CFI-UGO 90]
CFI Architecture, *"CAD Framework - Users, Goals, and Objectives"*, Version 0.92, 1990

[Chen 76]
Chen, Peter Pin-Shan, *"The Entity-Relationship Model - Towards a Unified View of Data"*, ACM Transactions on Database Systems, vol. 1, #1, March 1976, pp. 9-36

[DeMicheli 92]
DeMicheli, Giovanni; Ku, David; Mailbot, Frederic; Truong, Thomas, *"The Olympus Synthesis System for Digital Design"*, Stanford, CA 94305

[DeMan 92]
De Man, Hugo, *"Design Technology Research for the Nineties: More of the Same?"*, Proc. EURO-DAC92, pp. 592-596, September 1992

[Dimes 93a]
Dimes Design and Test Centre, *"DDL and DML for Meta Data Management in the Nelsis CAD Framework"*, Delft University of Technology, P.O. Box 5053, 2600 GB Delft, The Netherlands, December 1993

[Dimes 93b]
Dimes Design and Test Centre, *"The Nelsis CAD Framework: DMI Application Notes"*, Delft University of Technology, P.O. Box 5053, 2600 GB Delft, The Netherlands, December 1993

[ECMA 91]
National Institute for Standards and Technology, *"Reference Model for Frameworks of Software Engineering Environments"*, Draft Version 1.5, Gaithersburg, Md., 1991

[EDIF 88]
Electronic Industries Association, Engineering Dept., Paul Stanford (ed.), *"Electronic Design Interchange Format"*, Version 2 0 0, ANSI/EIA-548-1988, Recommended Standard EIA-548, March 14, 1988

[Fiduk 90]
Fiduk, Kenneth W.; Kleinfeldt, Sally; Kosarchyn, Marta; Perez, Eileen B., *"Design Methodology Management - A CAD Framework Initiative Perspective"*, 27th Design Automation Conference, 1990, pp. 278-282

[Gajski 83]
Gajski,D., Kuhn,R.H., *"Guest Editors Introduction - New VLSI Tools"*, IEEE Computer, vol. 16, #2, 1983, pp. 14-17

[Gedye 88]
Gedye, David; Katz, Randy, *"Browsing the Chip Design Database"*, 25th IEEE/ACM Design Automation Conference, 1988, pp. 269-275

[Graver 93]
Graver, Justin O., *"T-gen User's Guide"*, Computer & Information Sciences, U of Florida, E301 CSE, Gainesville, FL 3611, 1993

[Greiner 93]
Greiner, Alain; Pecheux, Francois, *"ALLIANCE: A complete Set of CAD Tools for teaching VLSI Design"*, Universite PARIS VI, 4, Place Jussieu 75252 PARIS Cedex 05 FRANCE, 1992

[Grosch 91]
Grosch, Josef, *"Puma - A Generator for the Transformation of Attributed Trees"*, technical report 26, Gesellschaft für Mathematik und Datenverarbeitung mbH; Universität Karlsruhe, November 1991

[vdHamer 94]
van den Hamer, Peter; Lepoeter, Kees, *"Managing Design Data - The 5 dimensions of CAD Frameworks, Configuration management and Product Data management"*, Philips Research, Eindhoven, NL, doc# JCF/Philips/001-1/7-Nov-94, November 1994

[Harrison86]
Harrison,D.S.; Moore,P.; Spicklmier,R.L.; Newton,A.R., *“Data Management and Graphics Editing in the Berkeley Design Environment”*, ICCAD 86, Electronics Research Laboratory, University of California at Berkeley, 1986, pp. 24-27

[Heusinger 93]
Heusinger, Peter; Ronge, Karlheinz; Stock, Gerhard, *“Moderne Entwurfswerkzeuge - aber bloß welche (1. Teil)”* (in German) #2, 1993, pp. 86-111

[Hunzelmann 92]
Hunzelmann, Uwe; Wilkes, Wolfgang; Schlageter, G., *“Design of a Tool Interface for Integrated CAD-Environments”*, European Design Automation Conference 1992, pp. 558-563

[Hüwel 92]
Hüwel, Gerd, *“Top-Down-Design mit VHDL”* (in German), Elektronik, pp. 84-88, Franzis-Verlag, München, September 1992

[JCF 94a]
Krannich, Günter (ed.), *“JESSI-COMMON-FRAMEWORK V4.0: Global View - Detailed Functional Specification”*, SP2 Draft Version, JCF/SNI/???-01/29-Apr-94

[JCF 94b]
Task force *“Cadence Integration”*, JCF internal report, 1994

[Johnson 78]
Johnson, Stephen C., *“Yacc: Yet Another Compiler-Compiler“*, Bell Laboratories, Murray Hill, NJ 07974, 1978

[Kathöfer 90]
Kathöfer, Th.; Fox, W.; Nolte, D.; Pielsticker, K.; Quester, R.; Rupprecht, F.; Schrewe, M., *“A Database Interface for Phased Tool Integration”*, EuroDAC’90, pp. 24-29

[Kathöfer 92]
Kathöfer, Thomas; Miller, Julia, *“The JESSI-COMMON-FRAMEWORK Project - Subproject Development”*, in: T. Rhyne ed., Electronic Design Automation Frameworks, Elsevier Science Publishers B.V. (North-Holland), 1992

[Katz 86]
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

[Katz 87]
Katz, Randy; Bhateja, Rajiv; Chang, Ellis E-Li; Trijanto, Vony, Gedye, David, *“Design Version Management”*, IEEE Design and Test of Computers, vol. 4, #1, February 1987

[Kupitz 92]
Kupitz, Elisabeth; Tacken, Jürgen, *“DECOR - Tightly Integrated Design Control and Observation”*, IEEE International Conference on Computer-Aided Design (ICCAD), Santa Clara, 1991, pp. 532-537

[Lesk 75]
Lesk, M. E., *“LEX - A Lexical Analyser Generator”*, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975

[Linux]
NFS-Server implementation within the Linux operating system,
URL=ftp://ftp.sunsite.edu/pub/linux

[McLennan 92]
McLennan, Michael J., *“[incr Tcl] - Object-Oriented Programming in Tcl”*, AT&T Bell Laboratories, 1247 S. Cedar Crest Blvd, Allentown, PA 18103
URL=ftp://ftp.aud.alcatel.com/tcl/extensions/itcl-1.5.tar.gz

[Meersman 94]
Meersman, C., *“The VHDL Standard”*, European Space Agency Contract Report, E2S Software Engineering, Technologiepark 5, B-9052 Zwijnaarde
URL=ftp://aixesa2.estec.esa.nl/pub/vhdl/VHDLReport.ps.Z.uue

[Mentor]
Mentor Graphics SupportNet
URL=ftp://supportnet.mentorg.com/pub/mentortech/translators/utilities/

[NFS]
Nowicki, Bill, *“NFS: Network File System Protocol Specification”*, Request for Comments (RFC) 1094, Sun Microsystems, Inc., Mail Stop 1-40, 2550 Garcia Avenue, CA 94043, March 1989
URL=ftp://ftp.internic.net/rfc/rfc1094.txt

[ObjectDesign 94]
Object Design, Inc., *“ObjectStore Technical Overview*”, Release 3, March 1994
URL=ftp://ftp.odi.com/pub/docs/techsum.{framemaker,postscript}

[OMG 90]
Soley, Richard Mark (ed.), *“Object Management Architecture Guide - 1.0”*, OMG TC Document 90.9.1, Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701, 1990

[Ousterhout 90]
Ousterhout, John K., *“Tcl: An Embeddable Command Language”*, Proc. USENIX Winter Conference, 1990

[Ousterhout 91]
Ousterhout, John, *“An X11 Toolkit Based on the Tcl Language”*, Proc. USENIX Winter Conference, January 1991
URL=ftp://sprite.berkeley.edu/tcl/tkUsenix91.ps

[Ousterhout 94]
Ousterhout, John, *“Tcl and the Tk Toolkit”*, Addison-Wesley Publishing,
ISBN: 0-201-63337-X, 1994

[Parikh 93]
Parikh, Sandip; Bushnell, Michael L.; Sienicki, Jim; Ganesh, Ramakrishnan, *“Distributed Computing, Automatic Design, and Error Recovery in the ULYSSES II Framework”*, European Design and Test Conference (EDAC), Paris, 1994, pp. 610-617

[Parr 94]
Parr, Terence John, *“An Overview of SORCERER: A Simple Tree-Parser Generator”*, technical report, University of Minnesota, Army High Performance Computing Research Center, February 1994
URL=ftp://marvin.ecn.purdue.edu/pub/pccts/sorcerer

[Pergande 93]
Pergande, Michael, *“Inside STF”*, #FZI.055.0, Forschungszentrum Informatik, Haid-und-Neu-Str. 10-14, D-7500 Karlsruhe 1

[Schettler 94a]
Schettler, Olav, *“Advanced Tool Encapsulation - Goals, Issues, and Technology”*, Poster and Demonstration, Workshop on Applied Framework Research, JCF Project, Karlsruhe, April 14-15, 1994

[Schettler 94b]
Schettler, Olav; Heymann, Susanne, *“Towards Support for Design Description Languages in EDA Frameworks“*, IEEE International Conference on Computer-Aided Design (ICCAD), San Jose, 1994

[Silva 93]
Silva, Mario J.; Katz, Randy H., *“Active Documentation: a New Interface for VLSI Design”*, 30th ACM/IEEE Design Automation Conference, 1993, pp. 654-660

[Smith 77]
Smith, J.M. Smith, D.C.P., *“Database Abstractions: Aggregation and Generalization”*, ACM Transactions on Database Systems, vol. 2, #2, 1977, pp. 105-122

[Spiby 93]
Spiby, Phil (ed.), *“STEP Part 11: The EXPRESS Language Reference Manual”*, WG5, doc# 14, release draft, 29 April 1993

[Synopsys 92]
*“VHDL - Brücke zwischen Systemanalyse und Siliziumdesign”* (in German) Synopsys Produktübersicht, in: CADS, Jahrgang 5, Heft 6, September 1992, VP-Verlagsgesellschaft, Herrenberg, 1992

[Synopsys]
*“Synopsys Online Documentation”*, Version 3.1, Synopsys, Inc. 700 East Middlefield Road, Mountain View, CA 94043-4033 USA

[terBekke 92]
terBekke, Johan H., *“Semantic Data Modeling”*, Prentice Hall International (UK), 1992

[terBekke 93]
terBekke, Johan H., *“Object Databases: structure and behaviour”*, TU Delft, report 93-58, Delft, 1993

[Thomas 83]
Thomas, Donald, E; Hitchcock III, Charles Y.; Kowalski, Thaddeus J.; Rajan, Jayanth V.; Walker, Robert A., *“Automatic Data Path Synthesis”*, Computer vol. 16, #12, December 1983, pp. 59-70

[Thomas 92]
Thomas, I.; Nejmeh,B.A. (HP, Innovative Software Practices), *“Definitions of Tools Integration for Environments”*, 1992, pp. 29-35

[ToolTalk]
*“Solaris OpenWindows - ToolTalk in Electronic Design Automation, A White Paper”*, Sun Microsystems, Inc., 2550 Carcia Avenue, Mountain View, CA 94043-1100 USA
URL=ftp://sunsite.unc.edu/pub/sun-info/white-papers/ToolTalk/ToolTalk_EDA.ps.Z

[Udell 94]
Udell, Jon, *“Componentware”*, Byte, 19 (5), pp. 46-56, May 1994

[Wagner 91]
Wagner, F.R.; Viegas de Lima, A.H., *“Design Version Management in the GARDEN Framework”*, 28th ACM/IEEE Design Automation Conference, 1991, pp. 705-711

[Walker 85]
Walker, Robert A.; Thomas, Donald E, *“A Model of Design Representation and Synthesis”*, 22th ACM/IEEE Design Automation Conference, 1985, pp. 453-459

[Walker 91]
Walker, Robert A.; Camposano, Raul, *“Introduction to High-Level Synthesis”*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1991, pp. 3-34

[Wasserman 90]
Wasserman, Anthony I., *“Tool Integration in Software Engineering Environments”*, in Software Engineering Environments: Proc. Int’l Workshop on Environments, F. Long, ed., Springer-Verlag, Berlin, 1990, pp. 137-149

[vanderWolf 90]
van der Wolf, Pieter, *“On the Architecture of a CAD Framework: The Nelsis Approach“*, EuroDAC’90, pp. 29-33

[vanderWolf 93]
van der Wolf, Pieter, *“Architecture of an Open and Efficient CAD Framework”*, PhD Thesis Technische Universiteit Delft, June 1993

[VHDL 87]
Institute of Electrical and Electronic Engineers, *“IEEE Standard VHDL - Language Reference Manual”*, IEEE Std 1076-1987, March 31, 1988

## Acronyms

| | |
|---|---|
| **ASCII** | American standard code for information interchange |
| **ALU** | Arithmetic/logic unit |
| **CAD** | Computer aided design |
| **CPU** | Central processing unit |
| **CFI** | CAD Framework Initiative |
| **CMU** | Carnegie Mellon University |
| **DBMS** | Database management system |
| **DBM** | Database management |
| **DDE** | Dynamic data exchange |
| **DDM** | Design data management |
| **DIM** | Design information management |
| **DMI** | Design management interface |
| **DML** | Data manipulation language |
| **DO** | Design object |
| **EBNF** | Extended Backus-Naur Form |
| **ECAD** | Computer-aided design for electronics |
| **EDA** | Electronic design automation |
| **ER** | Entity-Relationship model |
| **FAR** | Framework architecture reference |
| **Fortytwo** | The Answer to the Great Question of Life, the Universe and Everything |
| **IC** | Integrated circuit |
| **ID** | Identifier |
| **IPC** | Inter-process communication |
| **IPSE** | Integrated project support environmen |
| **ISO** | International Standards Organization |
| **ITC** | Inter-tool communication |
| **HDL** | Hardware description language |
| **JCF** | JESSI Common Framework |
| **JESSI** | Joint European Submicron Silicon Initiative |
| **MS** | Microsoft |
| **MUX** | Multiplexer |
| **NFS** | Network file system |
| **OLE** | Object linking and embedding |
| **ONC/RPC** | Open network connect / remote procedure calls |
| **OODBMS** | Object-oriented database management system |
| **OSF/DCE** | Open software foundation / distributed computing environment |
| **PI** | Programming interface |
| **PTB** | Parse-tree builder |
| **STF** | Structurer and flattener |
| **TCP/IP** | Transmission control protocol / internet protocol |
| **Tcl** | Tool command language |
| **Txl** | Tree transformation language |

## Appendix. Design example: The DP32 microprocessor

Throughout this thesis we use design descriptions from a hypothetical processor called the DP32 as running example. These descriptions are taken from the first edition of *"The VHDL Cookbook"* by Peter J. Ashenden of University of Adelaide, South Australia* [Ashenden 90]. The DP32 is a hypothetical 32-bit micro-processor with a simple instruction set. There are 256 general purpose registers, a program counter and a condition code register. The memory accessible to the DP32 consists of 32-bit words, addressed by a 32-bit word address. Tables 15-16 list the instruction set of the processor. Figure 31 shows its port diagram.

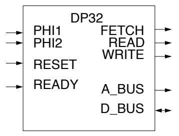


***Figure 31.*** *DP32 port diagram*

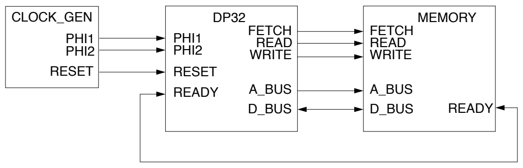


***Figure 32.*** *Test bench circuit for the DP32 microprocessor. The clock_gen component generates the two-phase clock and reset signal to drive the processor. The memory stores a test program and data. These behavioural models for these two components are connected in the structural description of the test bench.*

*. Mr. Ashenden kindly gave permission to use his microprocessor design in this thesis.

The actual microprocessor is embedded in a test bench circuit which is depicted in Figure 32. As this thesis is about CAD frameworks and not about electronic design, the detailed design description is not relevant to us. Examples taken from this description are used throughout the text to illustrate concepts. What is relevant, however, is the overall disposition of the design. Figure 33 depicts the composition hierarchy of the design description for the DP32 test bench circuit along with two configurations.

| Instruction | Name | Function |
|---|---|---|
| Br-ivnz | branch | if cond then $PC \leftarrow PC + disp32$ |
| Brq-ivnz | branch quick | if cond then $PC \leftarrow PC + i8$ |
| Bi-ivnz | branch indexed | if cond then $PC \leftarrow r1 + disp32$ |
| Biq-ivnz | branch indexed quick | if cond then $PC \leftarrow r1 + i8$ |

***Table 15.*** *DP32 branch instructions*

| Instruction | Name | Function |
|---|---|---|
| Ld | load | $r3 \leftarrow M[r1 + disp32]$ |
| St | store | $M[r1 + disp32] \leftarrow r3$ |
| Ldq | load quick | $r3 \leftarrow M[r1 + i8]$ |
| Stq | store quick | $M[r1 + i8] \leftarrow r3$ |

***Table 16.*** *DP32 load and store instructions*

| Instruction | Name | Function |
|---|---|---|
| Add | add | $r3 \leftarrow r1 + r2$ |
| Sub | subtract | $r3 \leftarrow r1 - r2$ |
| Mul | multiply | $r3 \leftarrow r1 \times r2$ |
| Div | divide | $r3 \leftarrow r1 \div r2$ |
| Addq | add quick | $r3 \leftarrow r1 + i8$ |
| Subq | subtract quick | $r3 \leftarrow r1 - i8$ |
| Mulq | multiply quick | $r3 \leftarrow r1 \times i8$ |
| Divq | divide quick | $r3 \leftarrow r1 \div i8$ |
| Land | logical and | $r3 \leftarrow r1 \wedge r2$ |
| Lor | logical or | $r3 \leftarrow r1 \vee r2$ |
| Lxor | logical exclusive or | $r3 \leftarrow r1 \oplus r2$ |
| Lmask | logical mask | $r3 \leftarrow r1 \wedge \neg r2$ |

***Table 17.*** *DP32 arithmetic and logic instructions*

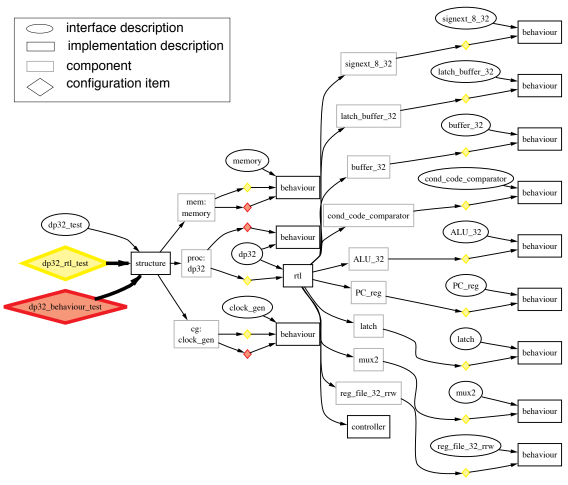


***Figure 33.*** *Structure of the design description for the DP32 test bench circuit. Two design configurations are shown. One, in which a behavioural description of the microprocessor is used; the other one which uses a structural description.*

# Acknowledgements

I wish to thank the following persons for contributing to this work:

My mentor, Prof. Dr. W. Gerhardt for approving this research topic, for copious technical discussions and for helping in the more administrative aspects of a promotion.

The members of the commission: Prof. dr. ir. P. Dewilde, Prof. dr. H. Koppelaar, Prof. dr. ir. F. Rammig, Dr. ir. R. van Leuken, and Dr. ir. P. van den Hamer for reading and correcting the concept version of this thesis.

My manager at GMD, Mrs. E. Abel, for making this research possible.

Dr. M. Sim, for initiating the contact with Prof. Gerhardt and lively discussing early ideas about the subject of tool encapsulation.

Dr. P. van der Wolf for discussing alternative approaches to extending the Nelsis schema and for giving me insight into the Nelsis messaging interface.

My colleagues at GMD, in particular A. Bredenfeld, R. Czech, S. Heymann, and A. Rockenberg for reading and discussing various versions of this thesis.

S. Heymann, for continuously encouraging me that tool encapsulation is in fact important.

The system administrators at the SET institute of GMD, R. Czech and B. Schwarz, for maintaining such a superb work environment.

Dr. D. Marquardt, for proof-reading my English.

My employer, A. Audi, for tolerating my continuous work on the topic of this thesis, yet unrelated to his business.

My wife, Birgit, for her patience.

## About the author

Olav Schettler was born in Hamburg on July 19, 1963 as son of the accountant Magrit Krämer and Heinrich Krämer, a school teacher for fine arts. He went through basic education in Wanne-Eickel, a town in the German Ruhr Area. After receiving the Abitur, he worked for three months in Blackburn, England, in the maintenance department of Dextralog, a company specialized in providing microprocessor based supervisory equipment for the shop floors of the weaving industry.

In 1982 and 1983, the author did military service on a ship of the German navy at Kiel. Upon discharge in 1983, he began studies of computer science at Dortmund University, which he finished with a degree as Diplom Informatiker in 1989, specializing in CAD for microelectronics. During the study, he also worked in the projects DACAPO-III and Desire at Lehrstuhl 1 of the Computer Science Department at Dortmund University.

In 1990, the author and his wife Birgit moved to Bonn where he joined the framework technology group headed by Mrs. E. Abel at GMD, Sankt Augustin. His tasks there included participation in the German DASSY project and the evaluation of the Jessi Common Framework (JCF). In 1991 and 1992, he participated in two integration projects conducted together with the CAD Framework Initiative, Austin, Texas.

In early 1993, the author left the evaluation subproject of JCF and joined the applied research subproject. During the summer, he spent three months in the Information Systems Group of faculty TWI at TU Delft, headed by Prof. W. Gerhardt. During this time, due to contacts with the Nelsis group at DIMES, some of the ideas underlying the work described in this thesis could be formulated. Back at GMD, he continued to work on the topic of design tool encapsulation. A first working software prototype of the new encapsulation framework service could be demonstrated at the Applied Research Workshop in Karlsruhe in April, 1994.

In October 1994, the author left GMD to work for Assem Audi + Co GmbH, a Meckenheim based, medium-sized software house successful in providing standard EDI solutions for financial institutions. Since February 1996 he works for METRO-net, a premium provider of online services in Germany.