AN INFORMATION-CENTRIC DESIGN EXPLORATION AND IMPLEMENTATION SERVER

by

Ole Bentz

Memorandum No. UCB/ERL M97/47

20 May 1997

AN INFORMATION-CENTRIC DESIGN EXPLORATION AND IMPLEMENTATION SERVER

by

Ole Bentz

Memorandum No. UCB/ERL M97/47

20 May 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

An Information-centric Design Exploration and Implementation Server

by

Ole Bentz

Doctor of Philosophy in Engineering

University of California, Berkeley

Professor Jan M. Rabaey, Chair

Advances in fabrication technologies will soon enable the integration of billions of transistors on a single chip. This high level of integration allows designers to create extremely powerful and complex systems on a single die. Systems consist of complex heterogeneous components, each of which may require significant expertise to design and optimize. Heterogeneity is found everywhere: the tools used to design each component may differ, the design methodologies vary, the optimization techniques are different, etc. In order to sustain increasing levels of productivity and lower design times, it is clear that a system design framework has to manage the heterogeneity.

This dissertation presents a new paradigm called "Information-centric Design." This paradigm offers a novel model for designer-CAD environment interactions and facilitates design exploration throughout the system design specification and implementation process. The information-centric paradigm requires encapsulation of design expertise and is capable of providing design aid to the level of the encapsulated design knowledge. A prototype of an information-centric CAD environment called the "Design Server" is also presented, and design examples illustrate the benefits of the information-centric paradigm.

[illegible]

Jan M. Rabaey
Committee Chairman

An Information-centric Design Exploration and Implementation Server

by

Ole Bentz

To my wife, Kelly

Table of Contents

List of Figures

Acknowledgements

First of all I would like to express my gratitude to my advisor and mentor, Professor Jan Rabaey. I have greatly appreciated his relaxed style, good humor, and that he always makes himself available to his students. Without his ideas, guidance, and feedback this project would have never come to fruition. I couldn't have found a better advisor.

I am grateful to professors Randy Katz (Computer Science Dept.) and Paul Wright (Mechanical Engineering Dept.) for their feedback to both my quals proposal and dissertation. Professor Katz has made some important contributions to the computer aided design field in the past, and professor Wright is making significant contributions to computer aided design in the mechanical engineering field. I would also like to thank Min-Yu Hsueh from Escalade for sitting on my qualifying exam committee. He offered some important insights from an industry perspective which helped bring focus to the project.

David Lidsky, my colleague for 4 years, deserves special mention. He has contributed to this project in more ways that I am even aware of, through sharing of his creative ideas, fruitful discussions, and critical feedback.

Thanks also to Sean Huang, Lisa Guerra, Renu Mehra, my colleagues and office mates, for all our interesting discussions about the Hyper software. Even though our schedules often didn't coincide (some people do like to work the "early shift"), we still managed to share a little in each other prelims, quals, and job search "blues". In addition, thanks to Jeff Gilbert for chats'n'coffee breaks. Good luck to all of you.

I made some great friends in Cory Hall. Their friendships have been invaluable to me, and have been a source of help, support and laughter. In particular, Henrik Esbensen, a fellow Dane, brought many moments of hilarious joking and jesting. His friendship really lightened up a work day, and he is the best head-gasket changer I know. Another good friend, Tim Kam, gave me tremendous moral support while I was preparing for the qualifying exam, and helped calm my nerves and relieve my stress. And thanks to David Cline, with whom I spent many a lunch hour talking about extracurricular stuff. I especially enjoyed all of David's hiking stories. I am also indebted to Eric Boskin for all his practical help and moral support during the first year of grad school. He made me feel very much at home in the department. Eric, and his wife Trudy, were also great neighbors in Albany Village, and I enjoyed the times when Eric and I got a chance to play basketball together.

I wish to thank my parents for their unconditional love for me. They raised me (and my brother and sister) in a home where I felt loved, secure, and where I could grow and learn at my own pace. In this environment they built a lasting foundation for my life, a foundation that has provided direction, stability, and peace in the midst of challenges and stressful times. I want to thank my parents for giving me life, and for giving me the spiritual, emotional, and mental ballast needed to live a happy, balanced and fulfilled life.

Last, and most significant, I would like to thank my wife Kelly. Kelly, without your encouragement and support I don't know if I would have made it. Thank you for your patience through the tough times, when school got more than it's fair share of my attention. Thank you for giving me plenty of reasons to have a "life" outside of school, and for the many fun and wonderful moments we have shared together. "A wife of noble character who can find?" I found one.

CHAPTER 1

Introduction

The rapid increase in the complexity of electronic systems demands new approaches to the design process. Since the 1960's there has been constant research activity in the area of utilizing computers to aid with the task of electronic systems design, and myriads of high quality, sophisticated computer-aided design (CAD) tools have been developed. At the same time, advances in semiconductor manufacturing technologies have enabled a doubling every two years of the number of transistors (the basic components of a chip) that can be integrated on a single chip. Today, the state of the art allows tens of millions of transistors to be included on one chip.

The rapid growth in chip complexities is outpacing the available CAD technologies. There is a large gap developing between what can be manufactured and what can be designed with current design tools in a reasonable amount of time (see Figure 1-1). Bridging the gap will require the emergence of a design infrastructure that can aid the design of complex systems.

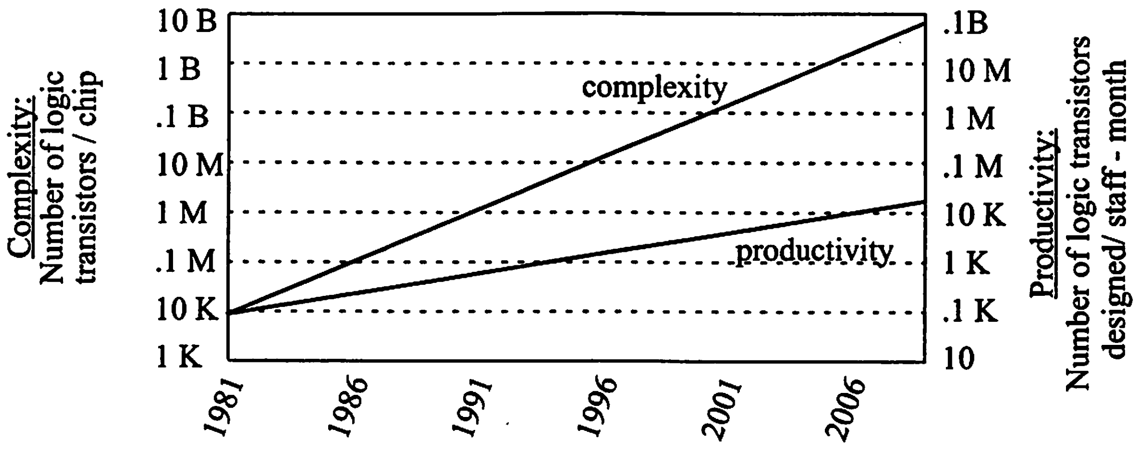


Figure 1-1. Complexity of designs vs. productivity supported by CAD tools [SIA96].

One of the main obstacles for the system design process is that systems bring together a broad range of design entities, each of which requires specialized expertise both to design and to integrate with other entities. It is impossible for designers to have both the breadth and the depth of knowledge required, so a design infrastructure must provide ways to capture design expertise and help designers exploit it during the design process.

This dissertation presents a new design environment for the exploration and implementation of electronic systems. The main novel feature of this environment is the way it encapsulates and uses design expertise to aid designers. The environment aids designers from the earliest pre-specification design stages, when exploration and information gathering is paramount, and it facilitates the implementation process as design specifications evolve.

This chapter first elaborates on the emerging gap between design technologies and chip complexities, then the aim and the contributions of this work are described. Finally an overview of the rest of the dissertation is given.

1.1. Problem Definition

In addition to the increasing complexity of the systems that can be built, there are tremendous pressures on the design process to meet shorter time-to-market constraints, as well as the economic pressures of a highly competitive global marketplace. Currently, design team sizes are growing to handle the increasing complexities, but that brings with it high personnel costs and the complexity of managing people.

To reduce development time and product life cycle cost, and to increase overall product performance, CAD tools must improve in two key areas, exploration and implementation. **Design exploration** is the process of evaluating the relative merits of alternate solutions to a problem. The goal of exploration is to provide information that can guide a designer in decision making. Variations of exploration include finding the best of a range of possible solutions, or finding one solution that adequately meets design requirements.

Exploration is needed to guide decisions at many stages of the design process. However, it is at the earliest stages of design (also known as the "conceptual" or "pre-specification" stages) that exploration has the greatest potential impact on the final design. Unfortunately, there is not a lot of factual data available during the early stages of design, so designers must rely on estimations or predictions. There are countless ways to obtain estimates, and finding the right technique to use in a given situation can be as difficult as using the technique. It is clear that aids must be provided to facilitate design estimation and exploration.

The current approaches to design exploration encompass a wide variety of methods that can best be characterized as "ad-hoc." From simple "back of the envelope" calculations, to elaborate spreadsheet computations, to consultation of data books, there is no generalized methodology for how to get quantitative information on which to base design exploration. In addition, the sources of information have varying degrees of reliability, ranging from the suspect "guess-timates" (a.k.a., "by the seat of the pants," "gut feeling," etc.), to the potentially competent "expert opinion," to factual data books. Other variations include the use of empirical models and estimator tools (i.e., CAD tools that have the ability to estimate). There are no generalized estimation techniques that apply to all areas of expertise, and even within a given area there are often several techniques that can be used under different circumstances. The heterogeneous nature of estimation methods, and the time-consuming task of finding and applying estimation techniques, prohibits extensive exploration. There is an obvious need for a more generalized and reliable approach.

The need for a more generalized approach also exists in the area of **design implementation**. Great progress has been made towards easing the design process by integrating large tool sets into CAD frameworks (CAD frameworks will be discussed more in chapter 2). Yet, there is still a myriad of tools and tool suites, and probably as many different specification languages. There are also many different implementation styles and target platforms that offer a spectrum of performance and cost trade-offs. A system design infrastructure must remove most of the burden of manipulating tools and design files, and of exploring alternative implementations, from the designers, so that they can be free to concentrate on the creative aspects of design. An infrastructure must also forge a connection between design exploration and design implementation such that the two tasks are not separate activities, but integral parts of a design process.

This dissertation presents an information-centric design environment that hides the heterogeneity of tools, files, and methodologies from the view of designers. The design environment is centered around information in the following ways:

A. It contains information: it encapsulates knowledge regarding specific design-related areas of expertise. This expertise is captured in such a way that computers can manipulate it and make it useful for designers who don't have the expertise. The design related knowledge includes the parameters and constraints that are relevant to a given field, definitions of the specification languages that are used, as well as models, estimation and implementation techniques, etc. The encapsulated knowledge can be distributed in nature, recognizing the need to draw on the expertise of different designers or organizations.

B. Its behavior depends on information: based on properties given by the designer (such as parameters, constraints, specifications, etc.) the design environment adapts itself to provide the best possible support. This is important, because design specifications evolve over time. A design infrastructure has to be tolerant to incompletely or gradually specified problems, and adapt its functionality to provide as accurate or detailed feedback to the designer as possible, considering the level of detail of the given design specifications. For example, there are many techniques that can be used to obtain estimates of power, so the design environment has to dynamically resolve which technique to use, based on all given design information.

C. It provides information: designers interact with the environment through intuitive "information request" commands. Some examples of typical commands a designer can use are "what is the chip area?" or "how fast is the chip?" In addition, the environment automates repetitive and tedious tasks to save time, thus facilitating the design process.

The information-centric design environment provides the necessary features to enable design exploration and implementation in a consistent fashion. The environment fills the gap between a designer and a heterogeneous set of design tools and methodologies, estimation techniques, databases, etc. (Figure 1-2). The aim of the design environment is to let a designer choose the specifications for a design, and aid with the rest, i.e., with the support of the entire design flow, including exploration and implementation.

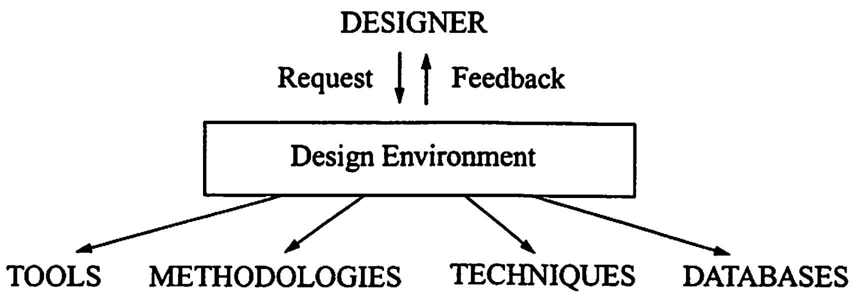


Figure 1-2. Information-centric Design Environment.

1.2. Contributions

The two main contributions of this work are an object-oriented knowledge encapsulation model and a design environment based on this model. The model specifies the contents of the knowledge objects and their interfaces, as well as the organization of objects into hierarchies. The design environment implements the necessary infrastructure and the capabilities that are required to exploit the encapsulated knowledge.

1.3. Dissertation Overview

This dissertation is organized as follows: Chapter 2 describes previous work in the CAD field, and outlines the many research results that formed the basis for this work. Chapter 3 studies the nature of the system design problem and outlines the many challenges that face system design environments. Chapter 4 describes the architecture of an information-centric environment, and chapter 5 discusses in detail each of the components of such an environment. Chapter 6 presents a prototype implementation of the design environment, and chapter 7 shows the use of the system through several design examples. Chapter 8 outlines directions for future work in this area, and conclusions are presented in chapter 9.

CHAPTER 2

BACKGROUND

The information-centric design environment has benefited greatly from the large body of research literature in the area of computer-aided design. This chapter gives a brief overview of this field, including some historical perspectives, and outlines the main results that have formed a foundation for this dissertation.

2.1. Foundational Work in CAD Frameworks

As the earliest solid state computers were being built in the late 1950's, and as they rapidly became bigger, faster and more complex, the idea of using existing computers to help design new computers became feasible. In 1968 IBM reported a computer-aided design system called GLEAM running on an IBM 1130 [Sass68][Sass71]. It aided with "design, simulation analysis and artwork generation for the fabrication of development-level Solid Logic Technology (SLT) circuit cards, boards, special circuits and other computer components" [Sass68]. Why is GLEAM not adequate for today's system design? The state of the art of semiconductor manufacturing was capable of producing about 500-1000 transistors on a chip in 1970. Designing for today's capabilities of greater than 10 millions transistors requires a much more powerful design approach.

During the 1970's and early 1980's, computers were being introduced into countless fields for design, optimization, verification, etc. Myriads of CAD programs and design description formats were developed to solve specific problems, but it wasn't until the late 1980's that work began on more unified approaches to managing CAD tools, data, and the design process. The following three sections discuss important previous work on the management of data, tools and design flows (or methodologies).

2.1.1. Data Management

The proliferation of design description formats created a new problem of how to enable various tools to share design data. In some cases, translation from one format to another would be adequate, but that solution is far from ideal in the general case. The communication between tools was greatly simplified by the introduction of design databases. In the OCT system [OCT89], a database was used to store all design data, and CAD tools had to be able to read from and write to that database. This virtually eliminated the need for translation of design data, at least for tools that were capable of accessing the database.

A need also arose for dealing with the evolving nature of design data. As designs evolved through refinements and improvements over time, it became important to manage several versions of a design. An example of a system that supports versions and configurations is the Version Server [Katz87].

The idea of representing data by reusable, extendable objects, as popularized by object-oriented (OO) programming languages such as Smalltalk [Goldberg83] and C++ [Stroustrup86], has become almost universally accepted. In most CAD systems it is common practice to represent design entities by design objects.

2.1.2. Tool Management

The primary aim of tool management is to make it easier for designers to use CAD tools. There are three main levels of management, with increasing degrees of ease of use. In the first level, ease of use comes from making all tools in a tool-suite access a common database format, e.g., the OCT database mentioned above. This eliminates the need for translation between various input and output file formats that otherwise would be required. The second level is to make all tools in a tool-suite adhere to a consistent invocation policy, such that command line flags are the same in the various tools. This eliminates the need to remember countless invocation options.

The first two levels require access to the source code for the design tools. In most cases that is not practical. The third level allows bringing a variety of tools together by using tool encapsulations. Tool encapsulations describe how to properly invoke a tool in a language that can be understood by computer programs, enable the use of generic graphical user interfaces, and are easily extensible when adding new tools. A prominent example is the Tool Encapsulation Specification which is a part of the CAD Framework Initiative's

(CFI) standard [CADF91]. This standard has been embraced by all commercial framework vendors.

As is the case with data management, object-oriented techniques are also commonly used to represent tools. The first use of OO tool representations was in Cadweld from Carnegie Mellon University [Bushnell89].

2.1.3. Design Flow Management

A significant number of CAD tools are required to perform electronic system design. Depending on the type of system, several hundreds of tools may need to be invoked. The order in which tools are sequenced is very important, and it is also paramount to run the proper tools on the right sub-problems, since tools tend to be highly specialized for a particular task.

A variety of strategies have been implemented to help manage design flows (also known as design methodologies). The strategies vary in scope and effectiveness, but they all offer some interesting or desirable qualities.

LAGER (University of California, Berkeley) [Shung88][Brodersen92]

LAGER is a silicon compilation system which provides design automation from high-level languages (C, Silage) or structural netlists, to layout and verification. The most common use of LAGER is to provide a netlist in the Structural Description Language (SDL), and let DMoct, LAGER's design manager, invoke all the necessary design tools.

The design manager executes a set of well established flows or design methodologies, and ensures that design tools are used at the right time, and with the right data in the design process. The design tools in LAGER include layout generators (for datapath and array structure tiling), a floorplanner, and placement and routing tools. All tools operate on an object oriented database, OCT [Harrison86]. The system uses the concept of libraries of hand-designed cells that can be reused, and includes parameterized cells that can be customized at compile time. Simulators are also available for verification at the netlist level (VHDL) or at the extracted layout level (switch level or Spice simulation). Some early work on Lager can be found in [Rabaey85][Ruetz86].

Ulysses (Carnegie Mellon University) [Daniell88]

Ulysses is one of the earliest VLSI CAD frameworks to offer support for design flow management. It uses a tool execution mechanism that is based on a blackboard model of rule-based systems. Predefined scripts specify high-level goals, and when scripts are invoked, a goal from a script is posted to the blackboard. Tools that can achieve the goal volunteer themselves, and the system lets designers choose which tool to use to accomplish the goal, according to certain performance criteria. Ulysses introduced the concept of automatic consistency maintenance between data, by keeping records of file dependencies, and being able to automatically update files when necessary.

VOV (University of California, Berkeley) [Casotto90]

VOV is unique in that it does not use predefined scripts of design flows. Instead, it observes a designer's activities and record them in a "trace". CAD tools are considered to be “black box” objects, and they have to report their activities to the trace facility. Traces are used both as documentation of the history of a design, as well as templates for future activity. VOV does not support hierarchical traces, so for even medium sized designs the flat traces can become unmanageable. The fact that no scripts need to be predefined makes VOV attractive.

The History Model (University of California, Berkeley) [Chiueh90]

The History Model combines the strengths of Ulysses and VOV. Design history is organized hierarchically, and a graphical interface is used to allow users to browse the history. It allows alternate design versions to coexist, and it can “restore” previous states of a design by a simple point-and-click operation.

Cadweld (Carnegie Mellon University) [Bushnell89]

The successor to Ulysses moved away from having specific tools hard-coded into the task descriptions. Cadweld made it easier to add new tools by using an object-oriented tool class hierarchy. However, it didn’t capture data in a similar class hierarchy, so problems still existed when existing tools changed.

Texas Instruments Flagship Design SystemTM (Texas Instruments) [Rumsey92]

The Flagship system is noteworthy for its use of hierarchical design flows, and for allowing alternative branches through a flow. Flagship is also capable of reducing the flow execution time by exploiting parallelism in the flow descriptions. Flagship supports various “desktop models”, which enables designers to choose which way the system should present their data, tools, and design flows depending on what is most relevant for their specific design scenario.

Odyssey (Carnegie Mellon University) [Brockman92]

The third generation CAD framework from CMU extended the object-oriented type hierarchies to include design data. This allowed dynamically binding a specific tool to a given task at run-time, based on the data type information. In the Hercules Task Manager [Sutton93], design flows are described in a schema of types and relationships between data and tasks. This, in effect, defines all the possible flows, without requiring a designer to follow a particular sequence. The specific sequences a designer performs are recorded as a history, and can be replayed at a later time. At run-time, Hercules can dynamically put together flows consisting of primitive tasks.

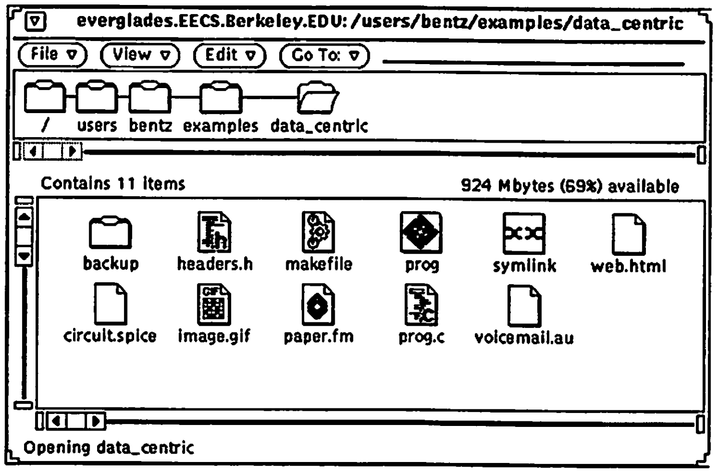
The Minerva Process Manager [Jacome92] provides a link between predefined flows and design objectives. It helps select and order design flows to achieve given objectives. Minerva introduced the notion of "domains", which this dissertation expands upon in Chapter 4.

2.2. Different CAD Paradigms

There are many different ways to approach the issue of how to present and manage data, tools and design flows in a CAD framework. Rumsey and Farquhar [Rumsey92] gave an excellent overview of three common approaches, and the next three sections are based on their descriptions. The fourth section describes the information-centric approach that is used in our design environment. The four approaches are essentially four different ways to look at the same problem.

2.2.1. The Data-centric Approach

The data-centric approach focuses on design data, typically stored in files or databases. Tools are either chosen based on the type of a data object, or are selected from a list of available tools. This approach is useful in design scenarios where a large number of data files are used to represent the design, such as in VLSI design, and where it is common for tools (e.g., a layout editor) to repeatedly operate on each of those files. In such cases, a data navigator can be adequate, such as the file manager shown in Figure 2-1. This interface is similar to what is used in the Apple/Macintosh user interfaces. The interface can display hierarchical relationships between data (in this case directory trees and contents), and allows users to navigate through the data and to point and click on the object on which to operate.



Contains 11 items

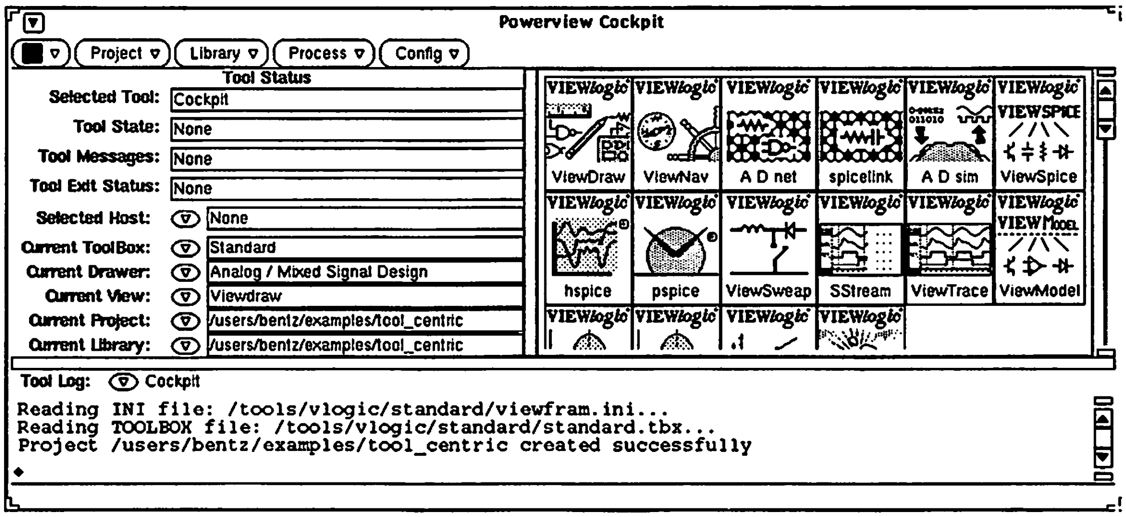
924 Mbytes (69%) available

Opening data_centric

Figure 2-1. The Data-centric Approach.

2.2.2. The Tool-centric Approach

The tool-centric approach focuses on a set of available tools and makes it convenient to invoke any of them. Users identify their design of interest, and subsequent tool invocations automatically operate on the given design. This approach is useful when many different design and analysis tools repeatedly need to operate on a single design file. One example is analog circuit design where a circuit is being designed, and simulation tools are repeatedly used to evaluate circuit performance. Figure 2-2 shows a tool-centric interface where the design is identified in terms of a project name and a "view" (the currently selected aspect of a design, such as schematic, behavior, layout, etc.). The right half of the interface is devoted to tool icons that provides convenient point and click access to a suite (also called a toolbox) of design tools.



Tool Status

Selected Tool:	Cockpit
Tool State:	None
Tool Messages:	None
Tool Exit Status:	None
Selected Host:	None
Current ToolBox:	Standard
Current Drawer:	Analog / Mixed Signal Design
Current View:	Viewdraw
Current Project:	/users/bentz/examples/tool_centric
Current Library:	/users/bentz/examples/tool_centric

Tool Log: Cockpit

```
Reading INI file: /tools/vlogic/standard/viewfram.ini...
Reading TOOLBOX file: /tools/vlogic/standard/standard.tbx...
Project /users/bentz/examples/tool_centric created successfully
◆
```

Figure 2-2. The Tool-centric Approach.

2.2.3. The Design-Flow-centric Approach

The Design-Flow-centric approach focuses on tasks in design process. A design project is identified in a manner similar to the tool-centric approach, but users issue commands that correspond to specific tasks instead of dealing directly with tools and data. This approach is useful in many areas of ASIC design, where significant portions of the design flow follows predefined patterns. Figure 2-3 shows a design flow interface which portrays the steps in a library-cell characterization process. After identifying a design, users can point and click on the box that represents the task they want to be done, and the design environment ensures that all the preceding tasks are also executed, if necessary.

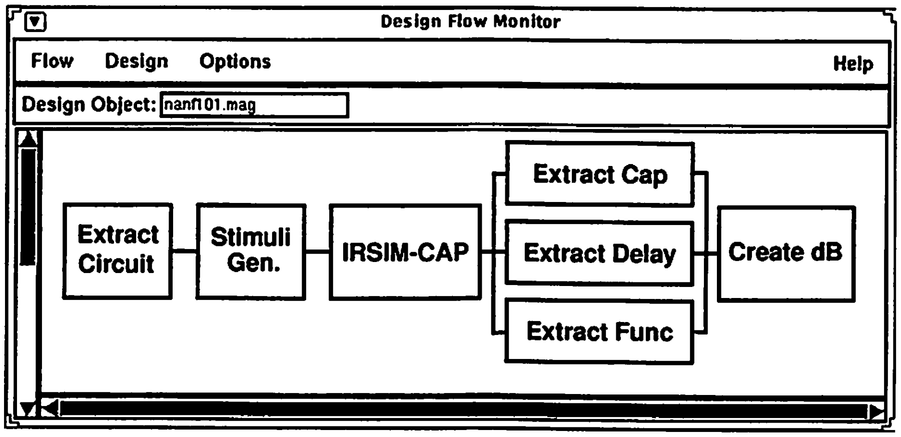


Figure 2-3. The Design-Flow-centric Approach.

2.2.4. The Information-centric Approach

The Information-centric approach focuses on the gathering and management of design-related information. This approach is useful in the design of complex electronic systems where relevant information is the key to making well-founded decisions about a design. The typical interactions between a designer and an information-centric design environment consist of specification, request, and feedback. The designer provides specifications for a design, either partially or completely, and requests information about some aspect of the design. The design environment responds by providing the best possible feedback based on the given specifications.

2.3. Other Related Work

There are other efforts in the field of CAD that are not readily categorized in the data, tool, or design flow areas. Yet, some inspiring ideas have emerged, and they will be discussed below.

2.3.1. Active Documentation.

Research in the area of electronic documents has given definition to thc terms compound documents [Mennella90][Birks90][Crossman92][Fanderl92] and active documents [Beach88][English90]. Compound documents, which have long been one of the *fortes* of Microsoft, consist of different parts such as text, spreadsheets, image, and multimedia content. Active documents additionally contain hypertext annotations or embedded scripts. Mario Silva introduced these concepts into the world of computer-aided design [Silva93,95]. In his work on the Henry system, he used active compound documents to facilitate the exchange of design data and documentation. Hypertext links could be included to provide links between related, distributed data, and scripts could be embedded to handle such tasks as installation of design files in a remote designer's file system. Each part of an active document was encapsulated in the standard MIME format [MIME92][Song95], so compound documents could easily be passed between designers by regular e-mail protocols, or by the Hyper Text Transfer Protocol (HTTP) used in the World Wide Web. Designs could also be submitted to tool servers across the Web for specialized services. Silva's work is representative of the new and growing field of Web-based design efforts.

2.3.2. Distributed Objects

Distributed objects has received a lot of attention in the past few years. Open standards have emerged to enable seamless access to heterogeneous, distributed data. The two main standards are the Object Management Group's Common Object Request Broker Architecture (CORBA) [Ben-Natan95] and Microsoft's ActiveX [Kauffman96]. In either case, objects can be physically stored on any networked computer and accessed through an interface that adheres to either standard's policies.

Also noteworthy is the Java language, which itself offers access to distributed, portable objects through the World Wide Web. Extensions to Java are also enabling Java programs to access objects through CORBA or ActiveX.

2.3.3. Utilization of Distributed Computational Resources

The use of remote resources for distributing the computational load of the design process has been considered to various degrees in CAD frameworks. One particularly nice example is the MMS CAD framework from MCC [Allen90,91]. It provides data management and encapsulations for tools and tasks, but the interesting feature is the utilization of heterogeneous networked resources. It uses a Process Control System (PCS) which monitors the load on a set of host computers. When tools can be executed in parallel, the PCS dynamically chooses lightly loaded remote hosts and dispatches the jobs. When each job completes, the PCS gathers the results and continues executing other tools in sequence. The dynamic selection of hosts makes it tolerant of host failures, and it significantly speeds up tasks that can be run in parallel.

2.4. Current CAD Support for Design Exploration

Design exploration is an important aspect of the design process. During exploration, data is gathered which enables choosing between alternative solutions. Exploration relies on the ability to predict the consequences of design decisions, because it is too costly to fully pursue all design alternatives to completion. There are a growing number of tools available that can help predict the outcome of certain decisions. They are typically very narrow in scope, and can only be applied to a limited set of applications. The next section describes a few examples of such "narrow" estimation tools, and the following section discusses two efforts that aim for more generalized exploration support.

2.4.1. Domain Specific Estimation Support

HYPER High Level Synthesis System [Rabaey91]

The Hyper suite of tools perform high level synthesis for DSP applications. The system takes as its input either Silage files or control/data flowgraphs (CDFG) describing an algorithm to implement. The synthesis tasks can be very time consuming, yet the time consumed is not always correlated to the quality of the result, so tools are provided to estimate various implementation costs in advance. All estimations rely on a set of pre-characterized hardware library cells and empirical models of interconnect and control. The estimator tools provide estimates of the following quantities:

- Area - quickly estimated by turning the complex synthesis problem into a "relaxed" problem. Minimum and maximum bounds of the implementation area can be estimated.
- Minimum Sample Period - Determines the minimum achievable sample period, measured in terms of clock cycles. This can be obtained from analyzing dependencies in the algorithm and determining the critical (longest) path.
- Minimum Clock Period - Determines the minimum clock period that can be used if all operations are to execute in one clock cycle. This can be determined from the delays specified for each hardware cell in the library.
- Power Consumption - estimated by assuming signals with white noise characteristics, combined with basic switching capacitances for each hardware cell in the library.

SPA (University of California, Berkeley) [Landman93,94,95,96]

SPA is a Stochastic Power Analysis tool that can estimate the power consumption of an architecture defined at the block diagram level. It requires a description of the architecture in terms of components and their connections, plus a symbolic description of the program executed by the architecture (if it is programmable). Actual input samples can be provided, or white noise inputs can be assumed. SPA uses a library of pre-characterized cells, estimates bus area and capacitance, and provides power estimates claimed to be within 20% of the results obtained with the orders of magnitude slower switch-level simulation using IRSIM-CAP.

Yoda (Carnegie Mellon University) [Dewey89]

Yoda focuses on the conceptual design of DSP filters. It allows designers to conduct initial feasibility studies, and it has an assistance subsystem that provides advice and performance predictions to aid the designer with decision making.

Pixie (Stanford University) [Smith91]

Spix (Sun Microsystems) [Cmel93]

Pixie and Spix both fall into the category of profilers. Pixie operates on programs for MIPS microprocessors, while Spix is for SPARC microprocessors. These tools allow existing programs to be monitored to collect dynamic instruction characteristics. They can be used to collect standard instruction-level profile information (such as op-code distributions, branch statistics, function usage, static and dynamic instruction counts), and in the case of Spix, users are permitted to develop custom instruction-level performance analysis tools (such as cache or pipeline simulators). These tools provide significant speedups over trace and simulation-based approaches. Profilers are useful to perform comparisons between different implementations of a program, and in locating pieces of a program that require optimizations.

Chip Area Estimators

Several efforts have been directed towards estimating the area of chips either during high-level synthesis (HLS) or logic synthesis. Estimates are calculated prior to performing time-consuming tasks such as scheduling (in HLS) or placement and routing in logic synthesis. Estimates are used both to inform designers about the likely outcome of time consuming tasks, and to guide synthesis tools (behavioral or logic) towards minimal area implementations.

For standard cell designs, the area of the cells is relatively easy to compute, but the area overhead introduced by the interconnect is more difficult. Some estimation approaches have been shown to yield results that are within 10% of the actual layout [Kurdahi86,89][Pedram89,91].

Area estimates have also been used in conjunction with delay estimates to help perform area-delay trade-offs. Some applications of this are to:

- minimize the total standard cell area subject to constraints on signal arrival times at the outputs [Ogawa90][Chaudhary95].
- synthesize a standard cell design and provide a 'companion' placement solution to guide a placement tool toward minimal wiring area and delay solution [Pedram91].
- use a cost function for routing contribution to chip area during logic synthesis to minimize routing area [Vaishnav95].
- RTL synthesis [Granacki83].
- perform high level synthesis and trade area for delay given desired performance or area

[Jain89][Parker91].

- synthesis of pipelined datapaths [Jain92][Park89].

2.4.2. General Estimation Support

While numerous application specific estimators exist, more generic estimation and exploration support is not as plentiful. This section describes three known efforts in this area.

Clio Prediction/Advice Environment (Carnegie Mellon University) [Lopez92]

Clio is integrated into the Odyssey framework, and works closely with the Design Process Manager, Minerva. Clio consists of two parts, a prediction subsystem that provides quantitative estimates, and an advice subsystem that provides qualitative information to help discriminate between available design options.

The quantitative prediction system is heavily linked to the Minerva process manager, and is tied into Minerva's transformational approach to the synthesis process. Thus, the prediction models aim at simulating the behavior of each synthesis step. The transformational approach is valid in many cases, but not all design scenarios can be modeled with this approach.

The qualitative design advice consists of static pieces of information collected from the entire design community, and each advice is tagged with a date, author, and keywords to help identify or show the validity of an advice. Designers can request advice related to a "design issue" (e.g., which layout style to choose: gate array, standard cell, or custom). For the chosen issue, a "discriminating factor" has to be identified (e.g., design area), as well as a subject (e.g., comparison). Clio will then search for relevant advice and present a list to the designer.

Unfortunately, a link between the two subsystems is not provided. It would be very useful if Clio would automatically search for relevant advice when a prediction indicates that a cost function has exceeded its acceptable bounds.

System Level Design Guidance (University of California, Berkeley) [Guerra94]

This work takes an algorithmic description of a system, extracts a set of properties that characterize the algorithm, predicts how fast it will execute on a certain processor, and guides a designer towards possible algorithmic optimizations. Although the underlying ideas are applicable in a broader sense, the implementation focused specifically on DSP algorithms.

PowerPlay (University of California, Berkeley) [Lidsky96]

PowerPlay takes an entirely different approach to providing design exploration support. It is not integrated into a CAD framework, but is freely available through the World Wide Web. Any modern web browser can act as a graphical user interface for PowerPlay.

PowerPlay uses a spreadsheet style approach to adding up implementation costs, in particular design power. Parameterized models can be entered, stored, and accessed by all users. The simplest models represent just a number (e.g., the power consumed by an LCD display is 3 Watts). More complex models require parameters (e.g., the area model of an adder requires the bit width of the two numbers to add). Global and local variables can be used to scale the results of individual models (e.g., supply voltage and clock frequency). Spreadsheets can be hierarchical (and parameterized), and are linked by "hyperlinks", i.e., a simple point and click operation brings to the foreground the desired spreadsheet.

The first version of PowerPlay relied entirely on models to obtain estimates. Later versions have added capabilities for executing analysis tools. Its main strength lies in the area of early conceptual level design space exploration.

2.5. Summary

CAD tools and frameworks have come a long way in the past 15-20 years. Going from no or limited use of computers in design to the use of full-blown frameworks has taken a lot of time. However, today's frameworks offer significant levels of data, tool, and task management and automation and enable designers to accomplish more and create designs with better quality than ever before. However, the levels of The normal case is to leave it running.complexity of designs that are on the horizon are so overwhelming that it is imperative that even greater levels of abstraction, encapsulation, and automation are provided. The next chapter discusses the challenges facing CAD environments in this area, and specifically the challenges for an information-centric environment.

CHAPTER 3

Challenges for Information-centric Design Environments

The external forces that drive the development of new approaches for design environments were discussed in chapter 1. They include demands for better management of the enormous complexity of designs, and a reduction of design cycles while yielding products with high performance to cost ratios. In this chapter we turn our attention to the internal restraining forces, the challenges that have to be overcome to meet the demands for better design aids. The next nine sections discuss the main challenges, and propose ways to overcome them.

3.1. Incomplete Specifications

In virtually all CAD tools it is assumed that a complete design description is available. If a description is slightly flawed, e.g., a semi-colon (;) is missing, tools generally will reject the description, perhaps after printing an error message. Significant effort goes into producing flawless design descriptions, and by the time an acceptable description has been crafted, the responsible designer has invested too deeply in the description to afford exploring alternative description approaches.

There is a clear need for CAD tool support for scenarios where complete descriptions are not available. This typically is the case at the earliest stages of design, where a lot of ambiguity and unresolved design issues exist. The design environment must be able to tolerate ambiguity and be able to provide assistance in spite of incomplete specifications. This will enable designers to explore a great variety of alternatives quickly and efficiently, before committing to the time consuming work of writing detailed design descriptions.

Another important observation is that design specifications change over time as more and more constraints are placed on a design. Design environments must allow gradual specification, and give the best aid possible at any time. This requires that environments are able to change their underlying behavior, while keeping a consistent interface, based on the designer's chosen specifications.

3.2. Heterogeneity

In the design of complex systems heterogeneity is found everywhere. To highlight a few examples, consider this list.

- Design Methodologies and Design Targets - various implementation strategies require different sequences of actions. Implementations can be targeted at a wide range of styles, for example standard cell, datapath, pla, FPGA, core processors, software, etc.
- CAD tools - require different input formats and invocation strings

- Design data - different access approaches are necessary depending on whether the data is in files, databases, or on Web servers.
- Estimation techniques - use countless different approaches, but are typically only valid in a narrow scope.
- Remote Servers - operating systems have different features

Heterogeneity obstructs the creative design process, because it is cumbersome for designers to deal with, and it gives designers frequent opportunities to make errors and mistakes. Therefore, it is imperative that design environments handle all heterogeneities in ways that are transparent to designers. The single key to this is **encapsulation**. By encapsulating entities, i.e., providing a computer-readable wrapper around them, a design environment can deal with these entities at a higher level of abstraction, where unified approaches are possible. Methodologies, tools, data, and estimation techniques can be encapsulated, as can the use of remote services.

3.3. Exploration and Trade-off Analysis

A large part of the early design exploration process deals with "what if?" questions. For example, "what if the chip is implemented in 0.25 μm technology?," "what if loop unrolling is applied to an algorithm?," "what if a color LCD display is used instead of a gray scale LCD?," etc. The design environment should make it convenient to ask such questions and get relevant feedback. In some cases it is also useful to make parameterizable automatic trial runs, or runs that are specified by a list of variations to try. The designer would specify the parameter bounds (e.g., let the clock period vary from 5 ns to

20 ns in 3 ns increments) or give a list of "issues" to change (e.g., structure of an algorithm, such as direct, transposed, or loop implementation), and choose one or more metrics to evaluate. In response, the environment would iterate through each of the given parameters or issues, and obtain the desired estimates.

Another variety of exploration is trade-off analysis. Trade-off analysis is concerned with how improving some cost metrics can be traded for a worsening of other cost metrics. In many cases this can be done by the type of parameterized explorations described above, where multiple parameters can be changed, and/or where multiple estimates can be obtained for each iteration.

In either case, a design environment should help manage the data obtained during design exploration. Results should be saved for later perusing, and be presented in an appropriate format, which includes tables and a variety of graph styles.

3.4. Estimation Accuracy And Traceability

Estimations, or predictions, form the foundation for the ability to perform design space exploration in an efficient manner. Without estimations, exploration would amount to pursuing design alternatives to completion, which is too time consuming. However, there is a major issue which has received very little attention in estimation tools and techniques, namely how reliable an estimate is. The guiding rule for when an estimate is reliable is that the design decisions that are being estimated should have a significantly greater impact on the metric under consideration than the estimation error.

To illustrate the importance of knowing how reliable an estimate is, consider this simple example. Suppose I ask "estimate the area of this chip" and the answer comes back: "2 mm^2". Before I can rely on this estimate I need to know the answer to some or all of these questions:

- What are the assumptions behind this estimate?
- How is it calculated?
- What is the margin of error, e.g., +/- 2% or +/- 100%?
- Is this accurate as an absolute number, or is it accurate relative to other similar estimates of other chips?

Over time a designer can build up confidence in the estimates made by a certain technique or tool. However, in the information-centric environment many different estimator techniques are employed at different times. The designer must be given the option to easily find out (trace back) how an estimate was obtained, and estimation tools should continue to evolve towards providing error margins or exposing assumptions.

3.4.1. Scaling, Interpolation, and Extrapolation

Estimation models and tools are typically designed for a narrow range of applications. As long as designs are within the covered region, estimates can readily be obtained. What happens if designers want to explore new territory? Usually, design environments provide little or no aid for such scenarios.

A powerful approach to extend existing estimations is to combine several estimates by using known scaling relationships or by interpolating between data points. Extrapolation based on estimates can also be used, but the results need to be treated with greater caution, since meaningless results can easily be generated in this way.

As an example of the use of scaling, consider VLSI circuits where there are well known scaling relationships when minimum feature sizes and supply voltages are scaled [Hodges88][Rabaey96]. Thus, if data is available for metrics such as area or delay in one fabrication technology, then scaling relationships can automatically be applied to find the expected values of the same metrics in another technology.

If scaling relationships are not known, but several estimates are available, interpolation can be used to obtain estimates between points. Some caution should be used, especially since the design space often is not a continuum, but rather a set of discrete points.

These methods can be used to leverage data from previous design experiences, i.e., previous designs or versions of a product. Being able to reuse previous experience in a way that is meaningful in the context of a current project is extremely valuable.

3.5. Command Overloading

Command overloading means that one command can have many different meanings. For example, consider the command "estimate power." There are many different ways to estimate power consumption, so this command has many possible interpretations. The design environment must be able to dynamically determine which technique to use, based on the all the specifications given by the designer.

Another aspect of command overloading is that the environment should be tolerant in interpreting commands. For example, if a command “power consumption” is available, and a designer asks for the “power” of a chip, the environment should either interpret the request to mean “power consumption,” or at least ask the designer if that is an acceptable interpretation. Thus, the command “consumption” could also be interpreted to have the same meaning. This in effect makes it easier for the designer to interact with the design environment.

If a command is interpreted to have one specific meaning, i.e., that one technique is identified that can satisfy the command, but the technique doesn’t succeed, the design environment needs to be able to detect this, and provide “fall back” schemes, such as looking for “second best” techniques, and either automatically executing them, or asking the designer to choose.

3.6. Reuse and Sharing

As systems become more and more complex, it becomes increasingly important to leverage previous design experiences. This can involve reusing complete designs or parts thereof, such as is commonly done with ASIC cell libraries, but it should also encompass reuse and sharing of design methodologies, previous results, etc. As mentioned in section 3.4.1. above, previous designs can be used in powerful ways in the exploration of new designs by using scaling and interpolation.

Another issue, namely the use of intellectual property (IP), is becoming increasingly important and is an area that needs significant efforts in the near future. As the concept of virtual corporations is becoming a realistic business model, the methods of selling and buying IP, and all the associated issues of protection of rights, and how to facilitate successful transfer of IP from one company to another, are becoming more and more pertinent issues. It is outside the scope of this dissertation to deal with most of these issues. However, it is clear that design environments need to play a significant role in the successful use of IP. The knowledge encapsulation model which is presented in this dissertation provides some answers to how design environments can play a big role in this area. A major recent effort in this area is the Virtual Socket Interface (VSI) [VSI96] standardization effort backed by dozens of companies. The main goal of this effort is to standardize interfaces of chip designs at different levels of abstraction, from register transfer level (RTL) to layout level.

3.7. Documentation

Documentation is necessary to describe features of designs, explain reasons for design decisions, and clearly define interfaces for a design. In many CAD frameworks documentation of designs amounts to tracking file dependencies and the history of actions. While this type of documentation is valuable, it is far from adequate, so designers are forced to handle the documentation process separately from the design process. In addition, when knowledge is encapsulated in the way that this dissertation proposes, it is even more important that documentation accompanies encapsulations to explain capabilities, limitations, identify the authors, etc.

Documentation needs to be an integral part of the design process. The environment must facilitate writing documentation, and retrieving documents that may be of relevance in a given circumstance. An integrated documentation strategy is also required to be able to deal with distributed documents, and to allow various users to add useful hints or experiences to existing documents. This raises many issues such as how documentation can be linked to designs or encapsulated knowledge, how to facilitate the documentation effort, where to store the evolving documentation if various users contribute, etc.

3.8. Distributed Resources

The traditional model for CAD tools and frameworks assumes that tools run on a local computer or compute server, and data is stored on a local file server. This model is being challenged by a distributed model, where data can reside anywhere in the networked world, and computation can be done remotely. Design environments must anticipate the challenges that arise in a highly distributed design world. The main goal is for design environments to exploit local and remote resources in a way that is transparent for designers. Since this area is related to this dissertation, but falls outside its scope, it will be discussed briefly below.

Distributed Storage - To provide seamless access to local and remote data, there are a number of issues that need to be resolved.

- data history - at least time stamps for last change of design data is required, to enable a design environment to determine when design data needs to be updated to be consistent with all other data in a design.

- consistency - there has to be redundancy built in to a distributed data model, to deal with network outages and remote host crashes. Also, it is desirable (for speed reasons) to have a caching scheme of remote files. Therefore, it is necessary to have a model that ensures consistency.
- security and integrity - since design data routinely is transferred across a network, it is necessary to provide a level of security appropriate to how sensitive the data is, and there has to be ways to ensure the integrity of the data transmission.

Distributed Services - In the distributed model, design tools can be executed either locally or remotely. One example of a remote design service is the IMADE lab in the mechanical engineering department of U.C. Berkeley [Sarma96]. Three-dimensional models of a casing or enclosure can be submitted to the IMADE lab electronically. In response, the lab's computer-controlled milling machine will fabricate a prototype of the case. Another service was described in [Silva93] which accepts a circuit design electronically and simulates it. Advances in electronic commerce are likely to bring similar remote commercial services into existence. At this time, it is not clear how remote services will communicate with customers, so we can only expect that each service will be different. To handle this heterogeneity, encapsulations are needed to be able to operate at a more uniform, abstracted level. Design environments must provide easy ways to write service encapsulations, by supporting most of the popular communication protocols, such as HTTP (hyper text transfer protocol), FTP (file transfer protocol), MIME (multipurpose internet mail extension), SMTP (standard mail transfer protocol), etc.

Locating Distributed Resources - When resources are scattered across a global network, the question arises how to find data and services. This is a large area of research that has received a great deal of attention in the recent years, especially in relation to the World Wide Web. The main approaches include

- Name Servers - up-to-date information about available services.
- Static Registries - centralized advertising locations (e.g., yellow pages). Not easily used by a design environment, unless they are structured in such a way as to enable easy parsing of contents.
- Local Database - a designer's or a company's preferred services.
- Search - an active scanning of remote locations to locate recently added services, e.g., new databases, data books, etc.
- Agents and Facilitators - special software agents that help facilitate a designer's requirements with the capabilities of a service.

WELD, a recent effort at U.C. Berkeley, is exploring the potential of the Web as a collaborative design medium. The goals of the WELD project is to provide a model and a standardized infrastructure for distributed design. The near-term goals are to create a distributed database to solve the problems surrounding distributed data storage, and a communication platform that facilitates designer-designer and designer-remote service interactions. Presumably, a future version of an information-centric design environment can be built on top of the WELD platform.

It should be noted that many of the challenges that arise in distributed design systems will be addressed by the emerging standards for distributed objects, CORBA and ActiveX. Thus, distributed design environments of the future are likely to be based on one of these standards.

3.9. Ubiquitous Access

The previous section outlined the requirements for a design environment to transparently manage the use of remote resources. It is also desirable to allow designers to access the design environment from remote locations. The traditional model of a team of designers sitting by their workstations in an office is gradually being augmented by other models. For example, designers may not be geographically close to each other, or designers can tele-commute (i.e., work from home or at generic offices not owned or run by the employer). These scenarios demand software to support collaborative work, which is outside the scope of this work. Another model is suggested by the InfoPad project [Barringer94], which uses a wireless multimedia terminal with pen and voice input. A design environment should be able to provide design support independent of where the designers are. Support should also be independent of the preferred type of interface, i.e., it should be possible to encapsulate the entire design environment into the designer's preferred tool or graphical user interface.

3.10. Summary

The challenges that an information-centric design environment have to overcome pose some serious demands, but also give opportunity and purpose for seeking out solutions that can effectively solve real-world problems. Thus, the demands for supporting incomplete specifications, encapsulating heterogeneity, and providing a framework for exploration and trade-off analysis that incorporates measures for estimation accuracy and traceability have helped bring focus to this dissertation and research project. Likewise, although not to as great an extent, the demands for reuse and sharing of designs and documentation in a highly distributed environment, as well as for ubiquitous access to the design environment, have shaped the goals of this work.

In the following chapter, the information-centric design model is described. This model was developed in response to the challenges outlined in this chapter, and provides a paradigm for the interactions between designers and CAD environments, as well as a model for organizing the internal components of CAD environments.

CHAPTER 4

SYSTEM ARCHITECTURE

The challenges discussed in chapter 3 are difficult or impossible to overcome using traditional CAD models. This chapter develops a new design environment model that addresses the challenges and defines the concept of information-centric design. The second half of the chapter (Section 4.2.) proposes a system architecture for an information-centric design environment.

4.1. Information-centric Design Model

There are many different ways to view a design environment. The three common paradigms are centered around data, tools or design flows, as discussed in Section 2.2. Each paradigm has its strengths and weaknesses, and each may be considered more useful at certain stages of design, or for certain designers. However, none of those paradigms are useful at the earliest stages of design, namely the conceptual or pre-specification phases.

A different model is needed which is capable of supporting conceptual design. The essence of conceptual design is to choose the specifications of a design, including specification of not only what to design but also guidelines for how to implement it. It is paramount that designers can gather the information they need to make well founded decisions. Thus, during conceptual design, designers want to interact with their design environment in terms of information.

The model proposed below is called the information-centric design model. It is centered around information in the sense that its behavior is based on given information (specifications, Section 4.1.1.), it encapsulates information (design expertise, Section 4.1.2. to 4.1.4.), and it is oriented towards giving information (feedback, Section 4.1.5.).

4.1.1. Specification Driven Model

The first property of the information-centric model is that the behavior of the design environment depends on the specifications given by a designer. If the design specifications are incomplete or loosely defined, the design environment provides aid and feedback that is appropriate to the level of detail of the specification. As the specifications gradually evolve towards being completely specified, the environment provides increasingly detailed and accurate aid or feedback. It is clear that a design environment cannot provide more specific help than what the given design specifications warrant. The notion that the environment adapts its behavior to provide the best possible aid at any given time is radically different from the traditional approach that requires exact descriptions before tools can provide any aid.

Example To illustrate the specification driven model, consider the following example. A designer has to determine the size of a multiplier for use in a VLSI design. The table below shows the design specifications that the designer gives, and the corresponding aid that can be provided for each case.

Specifications	Estimations	Other Aid
multiplier	**area** in units relative to other arithmetic cells. Assume (defaults): • simplest multiplier model (array) • 16 bits by 16 bits inputs • generic technology (λ undefined) • generic cell library	show available multiplier models, cell libraries, technologies
32 bits by 32 bits inputs	**area** in units relative to other arithmetic cells. Assume: • array multiplier model • generic technology • generic cell library	show available multiplier models, cell libraries, technologies
Choose specific cell library	**area** in units of λ	show available multiplier models, technologies
0.5 μm technology	**area** in units of mm^2	show available multiplier models
choose different multiplier models (booth, ...)	.....	

Notice that assumptions are made for loosely defined specifications. These assumptions use default models (e.g., array multiplier), default values (e.g., 16 bit wide inputs), or generic information such as the target fabrication technology or the choice of cell library. Therefore, in this scenario the specifications following "multiplier" can be given in any order, replacing the default or generic assumptions with the given specifications. This allows designers the freedom to choose design specifications in the order that seems most natural. Notice also that the specification driven model allows gradual refinement of the specifications, and provides area estimates in increasingly accurate and concrete terms. In addition to the specific estimates, the information-centric environment can also provide information to help the specification process.

To enable a design environment to adapt its behavior to given specifications requires that there is a way to capture what kind of aid or feedback can be provided in certain situations. This will be addressed in the next section.

4.1.2. Knowledge Encapsulation

To support the specification driven model it is clear that an information-centric design environment must have an underlying set of information resources that define which behavior the environment should take on, based on given specifications. In addition, the information resources must exist in various forms of refinement to correspond with the level of refinement of the given specifications. Figure 4-1 (a) shows a simplistic black-box model of the information resources. The information-centric environment uses the design specifications to identify relevant resources which in turn define the aid that can be provided by (i.e., the behavior of) the environment. Figure 4-1 (b) shows a more detailed picture of the information resources, where the resources are defined in templates, and the templates exist in various levels of refinement. Individual information templates are chosen, based on the given specifications, and together these templates define the aid that can

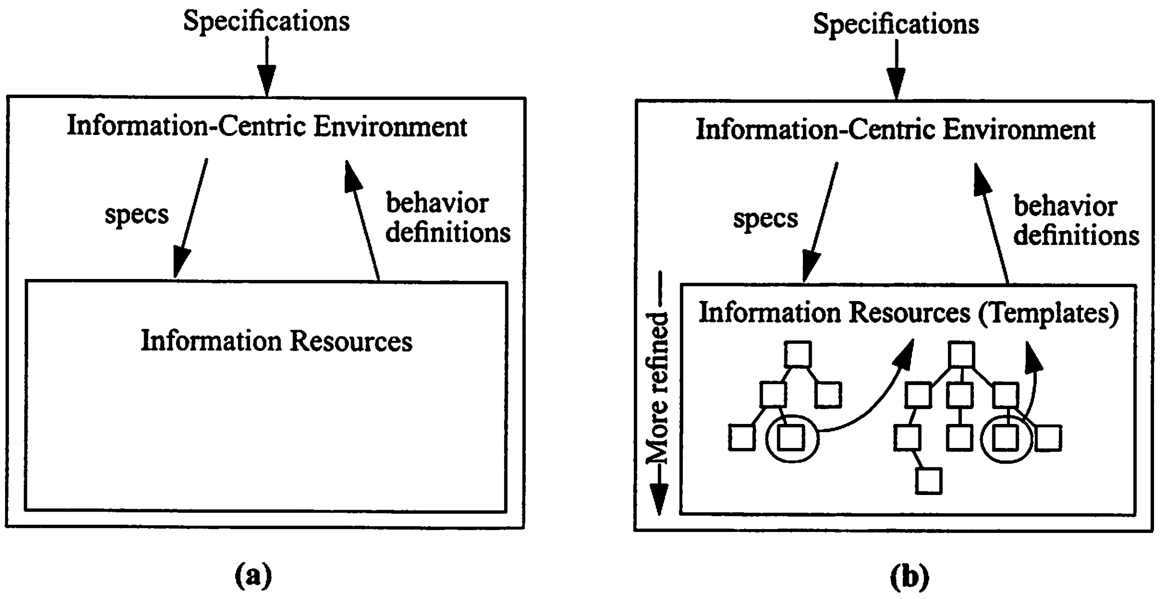


Figure 4-1. The Specification Driven Model
Requires a Set of Information Resources.

be provided.

In this dissertation, each information template is referred to as a "domain".

Definition:

A "domain" is an encapsulation of knowledge regarding a specific design-related area of expertise.

A domain is intended to capture the essence of an area of expertise, including common parameters and constraints, reusable design flows and tools encapsulations, and so on. A complete definition of the contents and interfaces of domains is given in Section 5.1.

The domain concept can now be described more formally using object-oriented (OO) terminology. A domain is an object class, and it has data structure (attributes), behavior (operations), and associations (see Figure 4-2). Domains have the following properties:

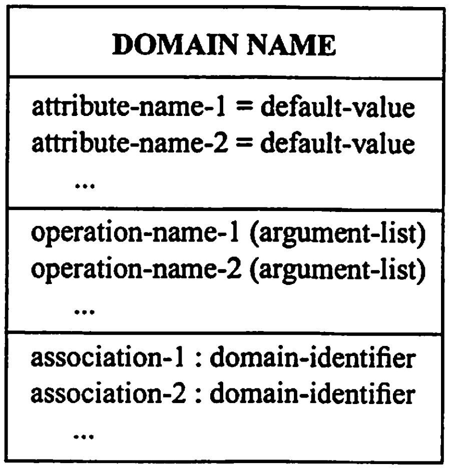

DOMAIN NAME
attribute-name-1 = default-value
attribute-name-2 = default-value
...
operation-name-1 (argument-list)
operation-name-2 (argument-list)
...
association-1 : domain-identifier
association-2 : domain-identifier
...

Figure 4-2. The Domain Class.

1. **Inheritance** Domains share attributes and operations with other domains through inheritance, based on hierarchical relationships. A domain inherits all the attributes operations, and associations from it's "super-class" domain (also called an ancestor), and adds its own unique properties, thus creating a more refined class that the super-class. Domains are only allowed single inheritance, i.e., it can only inherit one other domain. All domains that share a common ancestor are considered to be in the same "domain family."

   Example: Figure 4-3 shows an example of a domain class hierarchy. Notice that the domains lower in the hierarchy are the most refined.

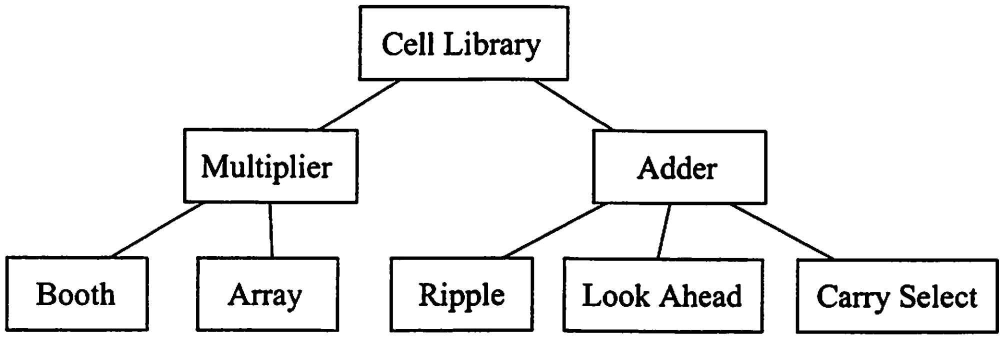


Figure 4-3. Example of a Domain Class Hierarchy.

2. **Polymorphism** The operations in domain classes are polymorphic, which means the same operation may behave differently in different domains. In addition, the same operation may have several different behaviors within one domain, based on the type of data on which it operates. Each behavior is implemented by a "method."

   Example: The operation (i.e., request for information) "critical path" behaves differently in a standard cell design domain and in a data flowgraph domain. In the standard cell case, the critical path is the maximum delay through a network of logic gates. In a data flowgraph, the critical path is the longest path (counted as the number of operations) in a flowgraph.

3. **Associations** Associations are used to establish relationships between domain classes. An association defines a unidirectional dependence of one domain upon another, i.e., upon the attributes and operations of the other. However, since all the domains that descend from an ancestor-domain share its attributes and operations, any domain that inherits from a given ancestor can be used to satisfy the dependence. This is somewhat similar to multiple inheritance, except that it is determined dynamically at run-time whether the domain identified by the association or any one of its descendants will be inherited.

   Example: Given the domains shown in Figure 4-4 below, the Adder domain can at run-time be associated with either the generic technology domain or any of its descendants. The choice will by default be the generic technology domain, unless the specifications point to one of the more specific technology domains.

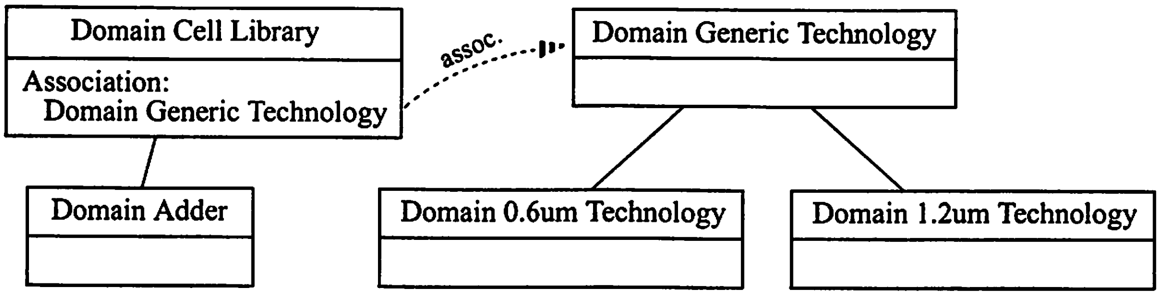


Figure 4-4. Example of Associations.

4. **Orthogonality** Domains (actually, entire domain families) that don't share a common ancestor domain are considered to be orthogonal. The concept of orthogonality is used to model the fact that designs have many facets or dimensions that are independent or only weakly dependent (cf. Gajski's behavior/structure/implementation Y-chart [Gajski88]). As an example, consider the facets "shape" and "material." The shape of an object is independent (with some exceptions) on the material it is made from. Thus, a hammer shape can be made from materials such as steel or plastic, resulting in very different objects. By keeping the "shape" and "material" characteristics in two different domain families (i.e., we consider them to be orthogonal) we are able to exploit the information in these domains independently of each other. Thus, we can later use a different shape domain (e.g., a wheel) with the same material domain (e.g., steel).

   The concept of orthogonality as it is used here may cause occasional ambiguities about which domain family should contain a piece of information. This is natural, since any abstraction is only a rough cut at reality; something will inevitably straddle the boundaries [Rumbaugh91]. In practice, however, a little trial-and-error usually clarifies the best place to include the information in question.

There are several reasons why encapsulation of design expertise is necessary. The four main reasons are given below.

- Enable greater reuse and sharing - not only designs can be reused but encapsulated design methodologies, techniques, and tools, can readily be reused. Encapsulations can also facilitate the documentation process by describing how to transparently access local and remote documents. In addition, scaling and interpolation relationships can be encapsulated to exploit data from previous designs in exploration of new designs.
- Enable the specification driven model - To enable a design environment to adapt its behavior to given specifications requires that the environment has knowledge about what kinds of aid or feedback can be provided at different stages of design and in different areas of design.
- Expand the scope of design - Designers are human and often have limitations in the breadth or depth of their knowledge. Encapsulated expertise can be used to extend the designers' abilities, by providing reasonable default values for design parameters and explanations for decisions that have to be made, etc.
- Hide heterogeneity - encapsulations enable a design environment to deal with a small set of uniform entities, instead of a large set of heterogeneous entities.

The key idea that sets this definition of domains apart from the approach used in the Odyssey framework [Jacome92], is that each domain captures a different "dimension" of the design space, such as fabrication technology, cell library, architecture, algorithm, etc. Since designs are constrained in many dimensions, we need to bring together several domains. The next section describes how domains can be combined in powerful ways to create a context for providing design-specific aid.

Summary This subsection described the underlying set of information resources in an information-centric environment. Each family of domains captures an area of design expertise, and each domain in a family represents a certain level of refinement. Domain families that don't share a common ancestor domain are considered orthogonal, i.e., relatively independent. By allowing associations, we can explicitly define when domains have dependencies on others.

4.1.3. Design Context

The information resources underlying the information-centric model were described above as orthogonal families of domains that represent various levels of refinement. An alternate approach would be to make very complex domains that would cover a very broad range of design expertise, instead of making the domains as simple and narrow in scope as possible. However, the orthogonal approach has the advantage of providing reusable building blocks that can be used to create the equivalent of complex domains on the fly by dynamically combining elements of expertise that are relevant to a given design. This section describes how domains are brought together to form the equivalent of a complex domain, called a "context," for designs.

A context can be defined as follows:

Definition:

A "context" is a union of orthogonal domains.

To continue the object-oriented terminology from the previous section, a context is a class that can have multiple inheritance of orthogonal domains and can dynamically change which domains it inherits at run-time.

Example Figure 4-5 (a) shows a context inheriting a single domain. Figure 4-5 (b) shows the same context at a later time in a design process, inheriting two domains. Notice that when multiple domains are inherited they must be orthogonal.

A context combines the attributes and operations of each of the domains it inherits. If there are attribute names that are defined several times across the set of inherited domains, only the first such attribute is kept (according to the order in which domains are inherited). Due to the orthogonality of domains this only occurs infrequently. Likewise, if there are

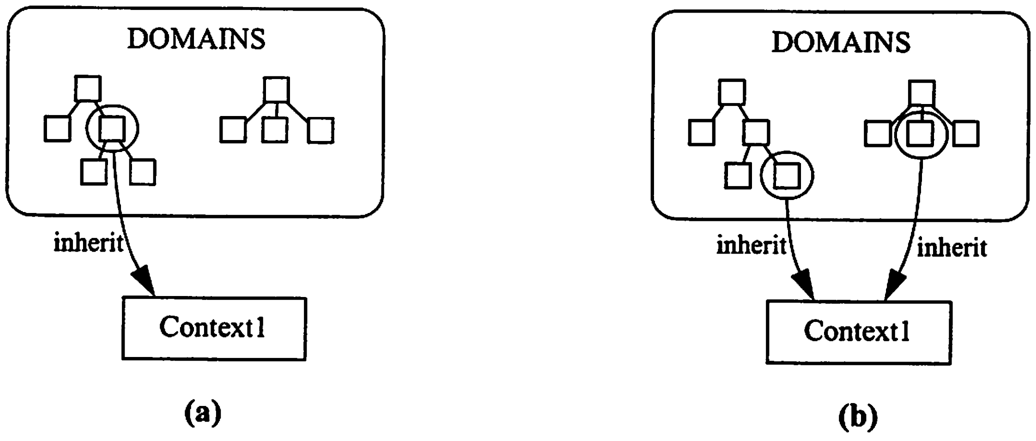


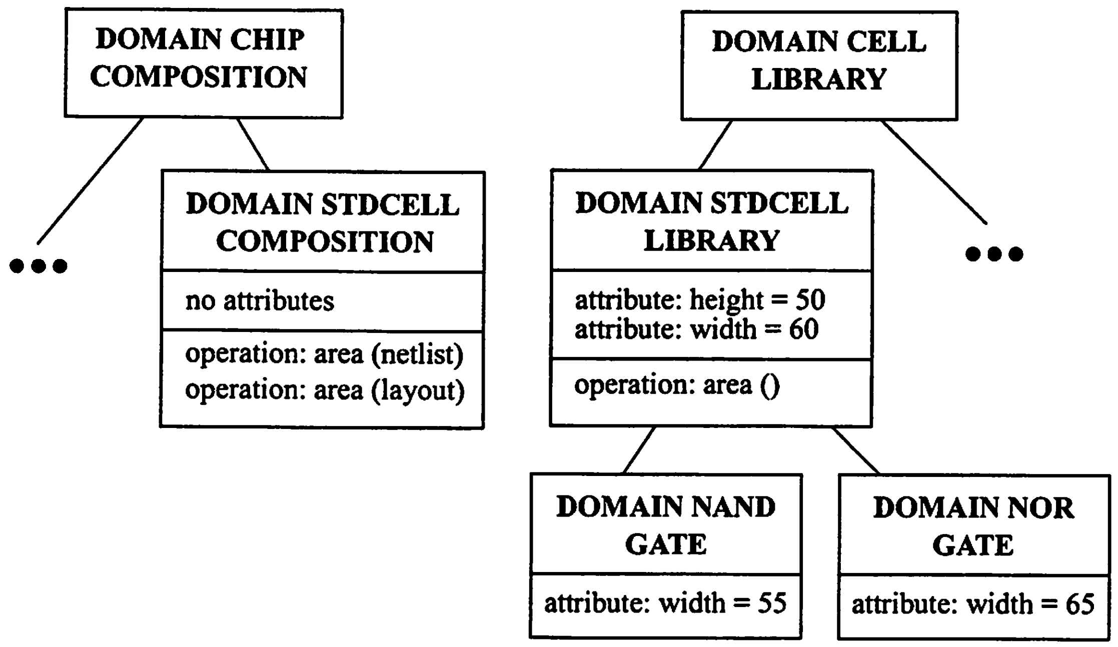
Figure 4-5. A Context Class can change its inheritance dynamically.

several implementations for the same operation with identical method name and argument list, then only the first method is kept. However, if two method names are the same, but their argument lists differ (i.e., they operate on data of different types), both are kept.

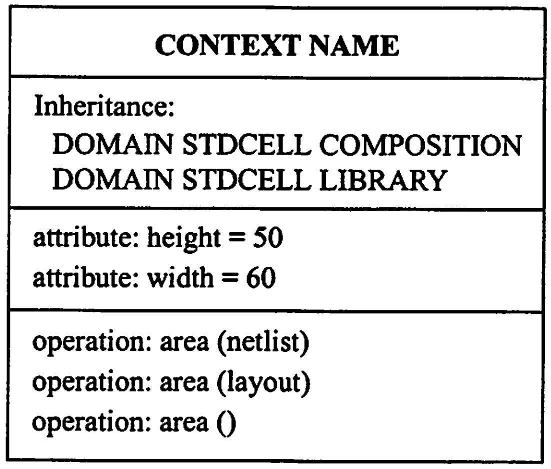
Example Figure 4-6 shows two domain families representing chip composition (i.e., information about how to combine components to create a chip) and cell libraries. The standard cell composition domain captures effects of combining standard cell components, and has one operation with two different implementations (methods) each operating on different types of data (netlists versus layouts). In the cell library domains, the standard cell library domain captures averages over the available cells (e.g., the average width of cells), as well as an operation that takes no arguments (i.e., it requires no data other than what is captured in the domain). The specific cells (nand gate, nor gate, etc.) contain the actual widths of these cells, and inherits the height attribute and the area operation. When a context is defined which inherits the standard cell composition domain and the standard cell library domain the context shown in Figure 4-6 (b) results.

When the inheritance relationships of a context change, the context has to ensure that the dependencies declared in domain associations are handled properly. Domains, by declaring associations, explicitly define their dependencies upon the attributes, operations, and associations of other domains. A context is therefore only meaningful when it inherits a given set of domains, plus all the associated domains. Section 5.2.1. discusses this in more detail and provides a set of rules for creating contexts.

Each design, or component of a hierarchical design, has an associated context. The



(a)



(b)

Figure 4-6. Context Example.

context can either be chosen by the designer, or it can be automatically chosen by the design environment, based on the given specifications (including parameters, constraints, and properties).

It is important to note that the context contains all of the knowledge that will be used to assist the designer. All of the tools that will be executed, all of the parameters that are relevant, all of the techniques and methodologies that will be used, have to be defined by the domains in the context. (In scenarios where this is too restrictive, new domains can easily be created by exploiting the object-oriented inheritance scheme and simply adding or altering some part of the domain knowledge). Therefore, the importance of choosing the context cannot be over stated. With a large collection of domains it is clearly not practical for a designer to manually find and select the right domains. On the other hand, while the environment can offer the help of a search facility that matches design specifications with the contents of domains, a task as important as selecting domains for a context should not be blindly left up to the design environment. The best approach is to allow for a balance of the two approaches, allowing the user to interact with search results whenever desired.

4.1.4. Design Objects

So far we have only discussed the object classes that are used to capture reusable design information. This section describes the objects that are used to capture actual designs. Before defining these objects, a brief review of domains and contexts will be useful. The domain class is used to encapsulate attributes and operations that are relevant in a specific area of expertise. The context class dynamically combines domain classes to form a complex set of information resources; it brings together the definitions of the aid that can be provided for a specific design scenario. The objects that are used to represent actual designs can now be defined as follows:

Definition:

A "design object" is an instance of a context.

A design object instantiates the context class. The instantiation does not cause a context to become unchangeable. The instantiated context is allowed to continue changing its inheritance, and any such changes are reflected immediately in the design object.

4.1.5. Information Gathering

An information-centric design environment exploits the design expertise that is captured in domains to help designers gather design-related information. The choice of which domains to include in a context (i.e., to inherit) determines the behavior of the environment, since commands are interpreted on the basis of a context. This section explains how the environment facilitates the context selection based on given design specifications, and how requests for information are handled.

The process of information gathering proceeds as follows (please see Figure 4-7).

- A designer provides the specifications for a design. The design environment searches for domains that are relevant to the given specifications and proposes one or more contexts. The designer chooses one of the contexts. Alternately, the designer can directly select a context by manually identifying the domains to be included.
- The designer *directly* requests the desired information, as opposed to the traditional approach of invoking a sequence of tools and extracting the desired data.

- The design environment interprets the request, based on the knowledge contained in the chosen context. As a result of the interpretation process, a specific script is chosen and is subsequently executed. (If there are two or more scripts that appear to satisfy a request equally well, the designer is asked to choose between the available alternative interpretations)

The behavior of the design environment changes when the specifications change. The change occurs in two places, namely the search and command interpretation stages. In the search stage, given different specifications, different contexts will be proposed. For general specifications, general domains will be found. More precise specifications will help locate more specific domains. In the interpretation stage, a command will take on a different meaning in a different context.

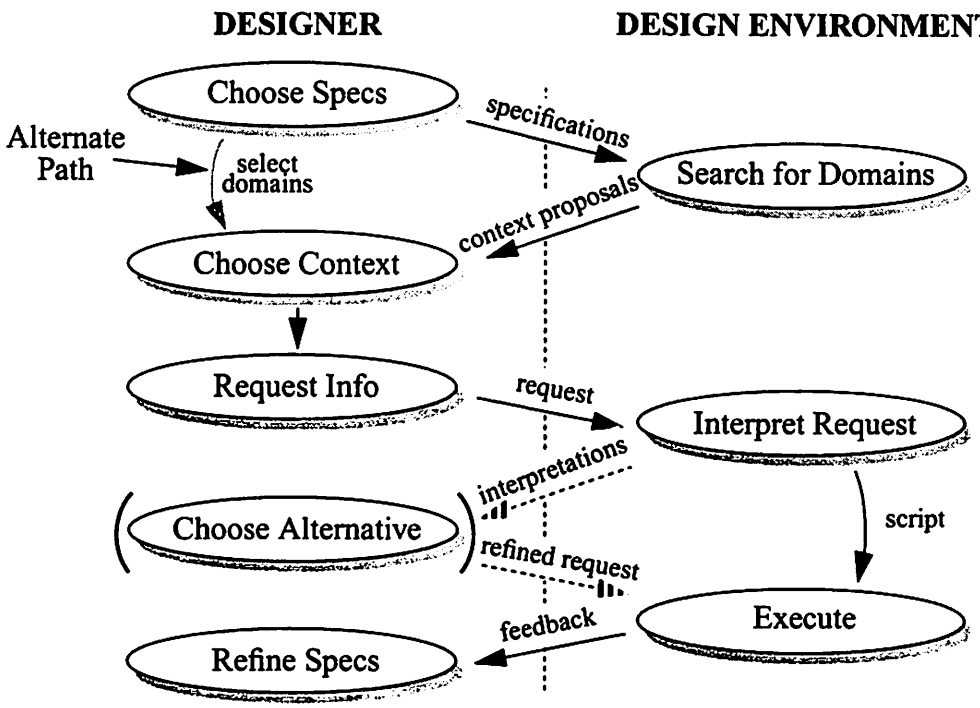


Figure 4-7. Information Gathering Process.

Example. Consider the design of a Fast Fourier Transform (FFT) filter for a real-time application. The FFT has to transform a 2048 point sample every 5 ms. Figure 4-8 below illustrates how a designer interacts with an information-centric design environment to determine if the FFT can be implemented on a Motorola DSP56K processor under the given constraints.

The example assumes that the design environment has a set of domains available that includes the domains shown at the top of the figure. The left column of the figure shows the state of the design as the designer takes the actions outlined in the middle column. The actions of the design environment are shown in the right column. The salient points of this example are:

- the designer's main task is to provide design specifications. The specifications chosen are either in the form of text ("FFT" and "DSP56K") or in the form of parameter values (points=2048).
- when a domain is added to the context, all its attributes become visible and its operations becomes available to the designer (provided there are no duplicate attribute and method names).
- the designer requests estimates for quantities of interest. The design environment performs all the necessary actions and computations, and provides feedback to the designer in terms of a result and possibly some details about the result (e.g., how it was calculated).

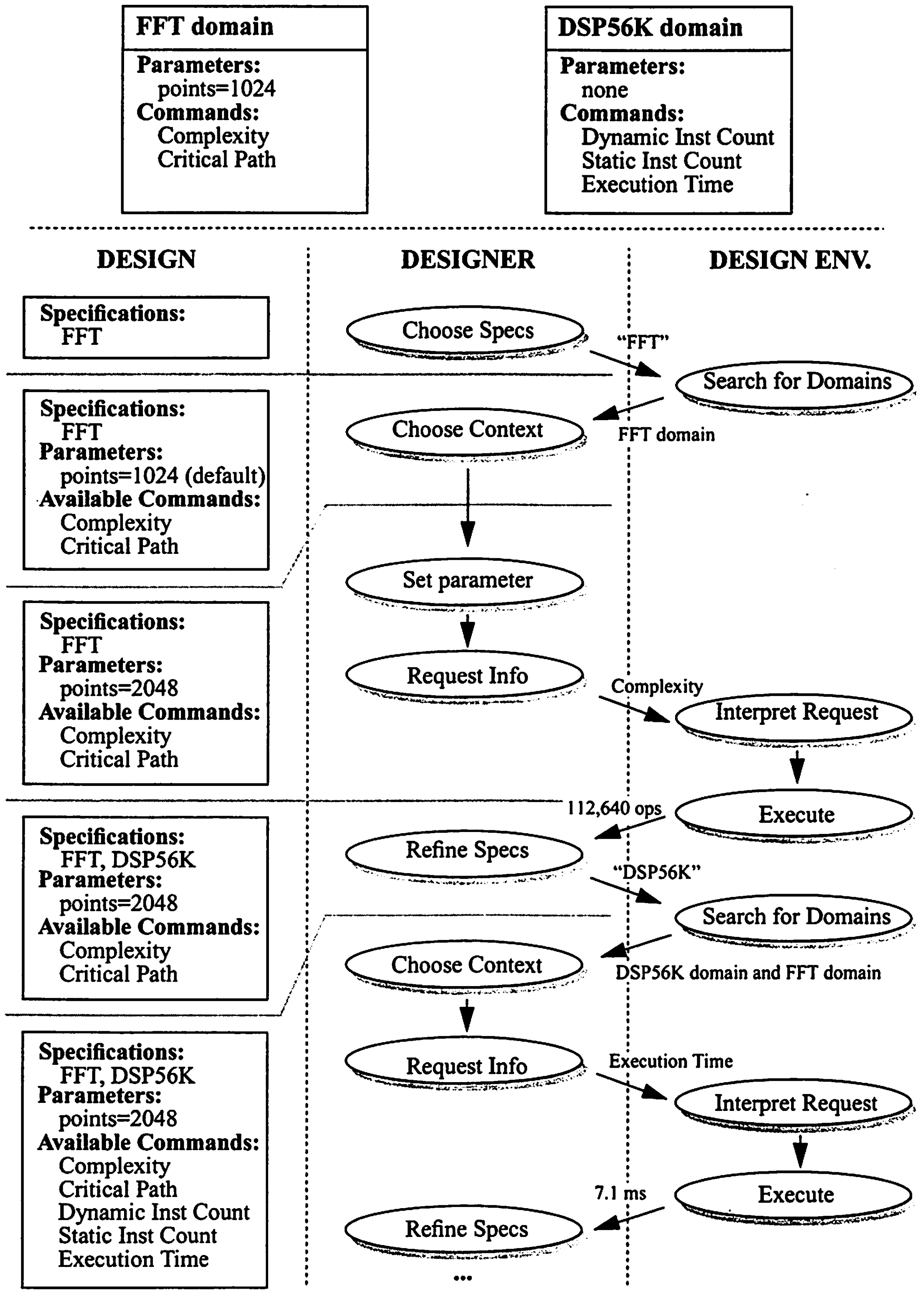


Figure 4-8. Information Gathering Example.

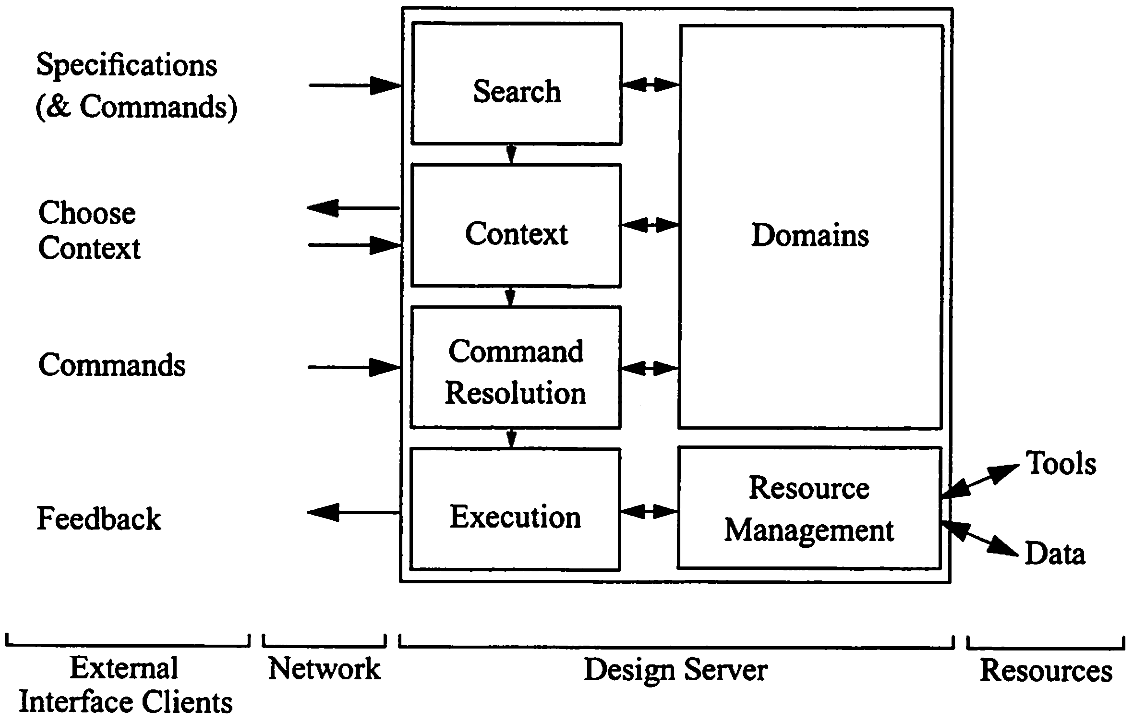


Figure 4-9. System Architecture.

4.2. System Architecture

The information-centric paradigm was discussed in Section 2.2 as being an alternative way of organizing CAD environments. However, an information-centric environment requires information resources, and their associated management, that are not needed in the traditional paradigms. Thus, to create an information-centric system we need to form a layer on top of the traditional CAD services (such as data, tool, and flow management), using the object-oriented methodology presented in Section 4.1. This section defines an information-centric system architecture and describes each of the subsystems in the architecture from the perspective of the services they offer.

Figure 4-9 shows a block diagram of an architecture that supports the information-centric paradigm. A designer provides design specifications in as much detail as possible. The environment in turn searches for appropriate domains, proposes one or more contexts and interacts with the designer to choose the context. The designer requests information or asks for an action to be performed. The given request is interpreted within the chosen context, and the environment resolves which technique to use to satisfy the request. The chosen technique is embodied in a script which is executed, and the result is returned to the designer.

As an alternative to the above interactions, the environment also allows a designer to directly choose a context without first giving specifications. After a context is chosen, the environment can also present the designer with a list of the requests that can be satisfied in the context.

To enable designers to access the design environment from anywhere in the networked world, an information-centric design environment has to adhere to a client-server model. The design environment acts as a server and designers access the environment remotely through external client interfaces. There can be several different interfaces to suit the particular needs of specific designers. User interface issues will be discusses in detail in Section 5.6.

The system architecture consists of six main subsystems (as shown in Figure 4-9): a search facility for locating relevant domains, a context definition/manipulation facility, a command resolution system, an execution environment, a domain management system, and a resource management that embodies the traditional CAD data, tool, and flow management. Each of these subsystems are described in detail below.

Search Engine The goal of the search engine is to propose one or more contexts that are as relevant as possible to a given design problem. The engine takes design specifications from the user and tries to locate relevant domains by matching the specifications with domain contents. It then composes one or more contexts from the domains that best matched the specifications, and assigns a figure of merit for each context. The figure of merit is computed from the number of exact and partial matches between specifications and information found in each context's inherited domains. The context(s) are finally proposed to the designer, who can choose any of the proposed contexts, or create an entirely different one using the context manipulation subsystem described below. The search engine is described in more detail in Section 5.4.

Context Manipulation The context subsystem facilitates context creation and manipulation. It allows a designer to choose between alternative contexts proposed by the search engine as described above, or it lets a designer add to or delete from a context's inheritance list.

Command Resolution The goal of the command resolution subsystem is to determine how to satisfy a given command or request. Command resolution gives one of three results:

1. A (method, arguments) pair. This occurs when a request only can be interpreted in one way in the given context. In this case the request needs no further refinement and can

be executed right away.

2. A set of (method, arguments) pairs. This occurs when a request can be interpreted in several different ways in the given context. In this case the request needs to be further refined by the designer.
3. No implementation for the given request. This occurs when a context has no method available to perform the requested function.

Resolution is based solely on the information contained in a given context, i.e., a request can only be resolved if an operation corresponding to the request is defined in the context. Operations can have several different implementations (methods), so the resolution process determines if there is one and only one interpretation available.

Command Execution The command resolution subsystem provides the execution environment for methods. Methods can invoke other methods, or can initiate a tool execution. The execution environment is described in detail in Section 5.3.2.

Domain Management The goal of the domain management subsystem is to provide convenient access to domain information to designers, the search engine, context resolution, and command resolution subsystems. This is primarily accomplished through appropriate interfaces to the domains. Designers may also view domain information, such as perusing method definitions or tool encapsulations.

Resource Management The goal of the resource management subsystem is to handle two of the traditional CAD services, namely data and tool management. This can either be handled internally in a relatively simple subsystem, or externally in established CAD frameworks. In the current implementation it is done internally and comprises file type identification and tool invocation.

4.3. Summary

This chapter has presented the conceptual model for information-centric design environments. This model places the focus on information; specifications, encapsulation of design expertise, and feedback. An object-oriented methodology is used to define the main concepts, domains and contexts. A system architecture was also presented, outlining the subsystems and the services they provide. The next chapter gives further definition to the basic components described in this chapter, namely domains, contexts, and the search engine, and also describes in detail the resource management system and the user interface.

CHAPTER 5

SYSTEM COMPONENTS

In this chapter the main components of the design environment are described in detail. The first section specifies the categories of design expertise that are captured in domains. The following section discusses contexts, and a set of rules are developed for creating and modifying contexts. In the subsequent section, an entity called a "design object" is defined, which is a representation of a design or a component of a hierarchical design. Design objects are instantiations of the context class, and they keep track of design files, handle command interpretation and execution, maintain a history, etc. The fourth section explains the search engine's mandate and functionality. The rest of the chapter is dedicated to describing the underlying resource management system and issues related to user interfaces.

5.1. Domains

In chapter 4 a domain was defined as an object class that encapsulates design-related knowledge from a specific area of expertise. This section gives a complete specification of the inheritance, associations, attributes, and operations of a domain. However, before getting into the details, it will be useful to review the overall organization of domains.

Domains are arranged in object-oriented hierarchies of classes, called families. Each domain is derived from one other domain, called an ancestor, and inherits all the knowledge contained in the ancestor. Thus, each derived domain adds a level of refinement, and only needs to define the knowledge that is not shared with the ancestor. Domains that don't share a common ancestor are considered to be orthogonal, which means they model different areas of design expertise. Therefore, several domain families are required, since each family is only intended to capture one area of expertise (one dimension in the design space).

Domains can be created by end users but are typically created by expert designers. Domains can be shared between designers, so the novice designer can gain access to the knowledge captured by experts by simply importing their domains. Domains can also be provided by design tool companies along with their regular software distributions, or by cell library companies along with their set of library cells.

The contents of a domain is defined in a text file. For the convenience of domain developers, a domain template, such as the one shown in Appendix C, can be used to outline the syntax for describing domain knowledge. The following discussion of domains is divided into four sections describing the inheritance, associations, attributes, and operations of domains.

5.1.1. Domain Inheritance

A domain is a part of a family that captures a specific design-related aspect. The family relationships are established by class inheritance: one domain inherits another domain by explicitly declaring the other domain as its ancestor. Inheritance implies that a descendent domain has exactly the same knowledge (attributes, operations, etc.) as the ancestor domain. However, the descendent domain can add to this knowledge, or override it, to create a more refined domain.

```
Syntax: Domain name -inherit other_domain {
    associations
    attributes
    operations
    }
```

```
Example: Domain Adder -inherit Cellib {
    ...
    }
```

5.1.2. Domain Associations

Domain associations are similar to inheritance in the sense that the knowledge captured in another domain in effect gets merged into the current domain. However, there are two key differences. The first difference is that inheritance establishes static relationships, while associations are used to establish dynamic relationships (at run-time). The second difference is that a domain association doesn't ensure that a specific domain is inherited; at run-time it will be determined whether the inheritance will be from the associated domain or from any one of its descendants. Recall, that all the domains that descend from an ancestor-domain share its attributes and operations; attributes and operations can be over-written or added to, but not subtracted from. Therefore, any domain that inherits from a given ancestor can be used in its place.

Domain associations enable a domain to identify dependencies on the information captured in other domains. Usually domain associations point to generic domains, or domains that provide default values. A special case is when a domain contains information that is only valid in the context of a certain other domain. An example of this is when a cell library domain contains delay measurements that are valid only for a certain fabrication technology. By listing the appropriate technology domain in the "associated domains" list, and by marking it with the "required" flag (-req), the design environment knows the conditions under which the measurements are valid. If the required technology domain is replaced by another, and if scaling relationships are also specified (e.g., how delays scale with minimum feature sizes) then the environment can automatically project the results to conform with the assumptions of the other technology.

Syntax: domains [-req] domain1 [domain2] ...

Example: domains -req Technology:Mosis:1.2um

5.1.3. Domain Attributes

Attributes and operations carry the information content of a domain. The operations contain the dynamic information, namely the methods that define run-time behavior. In contrast, attributes contain the static elements of the information, such as parameters, constraints, etc. For the purposes of capturing design-related information, simple (attribute name,default value) pairs are too limited. To provide a richer, more convenient description environment, nine categories of domain attributes are used, as described below.

Identifiers - a list of descriptive words that characterize the contents of a domain.

These words are used by the search engine to find domains that are relevant to given design specifications.

Syntax: key name [value1] [value2] ...

Example: key celltype shifter shift "right shift" "right shifter"

Paths - a list of paths along which to search for data that is relevant for this domain.

Paths are primarily used to locate default resources, such as file templates or design examples. For example, a technology domain can specify a path where SPICE simulation models are kept. A path can be specified as a uniform resource locator (URL), or as a path in a local file system.

Syntax: path path1 [path2] ...

Example: path /tools/cadence/share/library/low_power

Parameters - a list of common parameters and their default values.

Parameters are used to capture typical design parameters or variables. For example, an FIR filter domain has a parameter "NrOfTaps," and a technology domain has a parameter “lambda” (minimum feature size). A parameter’s default value is used for initialization purposes, but a designer is free to choose a different value at run-time. Parameters have an option called “keep” which designates whether a parameter value should be allowed to change if a domain that has the same parameter is added to a context. If the keep option is set, the value of a parameter will not be overwritten by the addition of other domains. As a general guideline, parameters that are typically chosen by designers should have the keep option set (e.g., the NrOfTaps parameter above), and other parameters (such as the parameter lambda above) should not have the keep option set, to enable swapping a variety related domains in and out during exploration. Parameters have three additional options: The “group” option allows parameters to be grouped together; the “type” option allows explicit declaration of the data type of a parameter (e.g., float, integer, or string) used for type checking; the “propagate” option specifies that a parameter’s value should be propagated to all subcomponents of a hierarchical design.

Syntax: parameter name -varname varname -value value [-keep] [-group menuGroup] [-type type] [-propagate]

Example: parameter “Number of Filter Taps” -varname NrOfTaps -value 1024 -keep -group “Filter Parameters”-type integer

Constraints - a list of typical constraints within a domain.

Constraints define when a design-related property is outside its limits. Constraints are used to notify designers when a property is determined to exceed its allowed values, such that the designer can take appropriate action. Constraint attributes can specify bounds on parameters or on generic design properties. A parameter constraint limits the values that can be assigned to a parameter; the constraint is checked every time a parameter is assigned a new value. A property constraint is used to check if the result of a method is within given bounds. The rationale is that a design environment only knows about design properties when they are extracted by a method. Thus, after a method is executed it is verified that the constraint is met (if there is a constraint declared for the result of that method). The "group" and "keep" options are the same as for parameters, and the "ignore" option lets a designer turn off checking of this constraint.

Syntax: constraint name [-group menuGroup] [-keep] [-ignore]

plus one of the following:

-parameter name -min value|-max value|-min value -max value|-values list_of_values

-method name [-units units] -min value|-max value|-min value -max value|-values list_of_values

Examples: constraint "Size" -parameter size -values {small medium large}

constraint "Voltage Range" -parameter voltage -min 1.0 -max 3.3

constraint "Maximum Area" -method area -type "mm^^2" -max 2.0

Tool Encapsulations - encapsulations of design tools.

A tool encapsulation is a definition of how to properly invoke a tool. It is used by a design environment to automatically run a tool, typically as part of a design flow. The tool encapsulation format used is similar to the TEF (Tool Encapsulation Format) from CFI [CADF91], but it adds two features: a list of computational resources where a tool can be executed (host names or names of a cluster of hosts), and an optional script for creating supporting files at run-time, such as simulation scripts or command files. Tool encapsulations have the following fields of information:

input arguments - a list of the arguments required by the tool. This includes files of certain types (e.g., a C file) and optional flags (e.g., "-a").

outputs - a list of the files that are produced by the tool. This usually relates the name of the inputs (files or flags) to the names of output files.

environment - the path and environment variables that have to be set up prior to invoking the tool.

invocation string - a string that describes a proper invocation, such as "toolname input_arguments input_files".

status information - definition of how to interpret a tool's status codes. For UNIX systems, and exit status of 0 (zero) means the tool ran successfully, and a non-zero value indicates a failure. Some tools use other exit codes to convey other messages.

host list - a list of host computers that can execute this tool.

preparation script - a script that at run-time creates supporting files before a tool is actually invoked.

Data Resource Types - a list of the types of data (usually files) of interest in this domain. Resource types are used by the design environment to distinguish between design representations. For example, a design can be simultaneously be represented by VHDL code, a schematic, and a layout. However, each of these resource types require different processing, so by specifying how to recognize the different types, the environment can automatically match methods that operate on a certain type with a design representation of that type. Each resource type is given a name (e.g., "c-file"), and a way to identify resources of that type is given, which can either be a name-based rule (e.g., *.c for C files) or a content-based rule (e.g., "look for the string '#!/bin/csh' at offset 0" for C-shell scripts). The content-based rule is similar to the UNIX "file" command which uses definitions in the "/etc/magic" file.

Unit Conversions - a list of conversion scripts for converting between common units. Conversions are needed when values with different units are compared or combined. The need arises when methods provide their results in different units. For example, one area estimation method may return its result in units of square millimeters, and another in square microns. In many cases the conversion scripts simply convert common units (e.g., millimeters to meters, or square inches to square centimeters). However, more complex scripts can also be defined, such as converting from power dissipation to heat. The standard prefixes (e.g., for kilo, milli, etc.) are built in. Conversions automatically occur when a method calls another and the result is returned in different units than what was expected.

Syntax: unit_conversion type1 type2 script_from_1_to_2 script_from_2_to_1

Example: unit_conversion Fahrenheit Celcius {[expr ($value-32)*5.0/9]} {[expr ($value*9.0/5)+32]}

Scaling, Interpolation, and Extrapolation - a list of scripts that define scaling relationships or interpolation/extrapolation data and/or techniques.

Scaling, interpolation, or extrapolation is needed to make reasonable predictions of data points that fall outside a known set of results. For scaling, one data point and an analytical scaling relationship need to be known to predict other data points (see Figure 5-1 (a)). For example, the results obtained from Hyper's database of low power library cells are valid for a 1.2 μm fabrication technology. When these results are used in a context where a different technology is used (e.g., a 0.6 μm technology), the results have to be scaled to make sense in the given context. Scaling relationships can be multidimensional, i.e., the effect of several different variables can be accounted for in a scaling relationship (e.g., the effects of both the minimum feature size of a chip and the supply voltage can be combined in a scaling relationship). For interpolation and extrapolation, a set of data points (typically empirical) are needed (see Figure 5-1 (b)). Inter/extrapolation also allow prediction of many variables, but this requires a multidimensional data set. Extrapolation can yield highly suspect results, so caution should be used, and designers should be notified when results have high error margins. Currently, only linear interpolation is supported (i.e., between two points), and extrap-

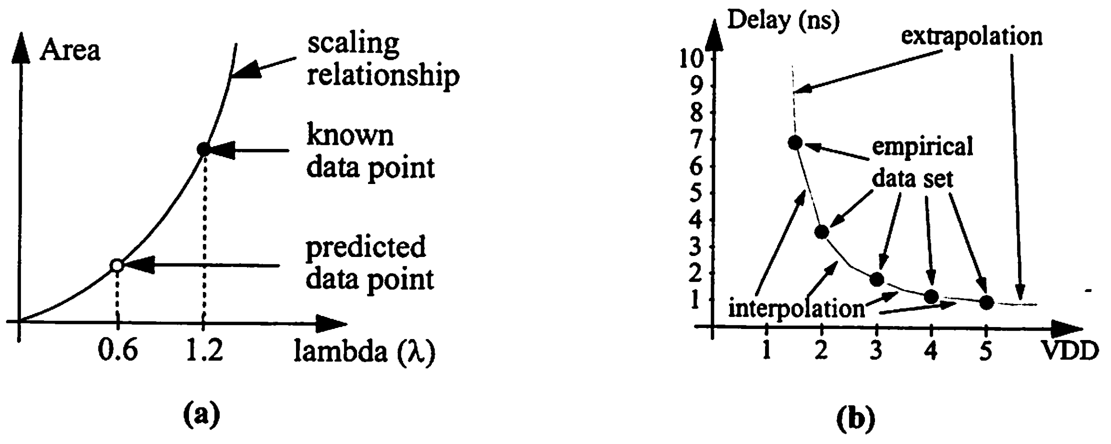


Figure 5-1. Scaling and Interpolation/Extrapolation.

olation is done by extending a line through the two closest points in the data set to obtain the desired data point. Future improvements may revise this to allow a wider range of interpolation/extrapolation schemes.

Syntax: translation -scale|-polate label script

Examples: In the following examples, please note the comments (on lines that begin with a '#' character). In the first example, the variable *value* is automatically set to the area value that needs to be scaled. The script defines an analytical scaling relationship that uses two lambda values (lambda - half the minimum feature size on a chip), the lambda which was assumed when the area was originally computed, and the lambda of interest. Since *lambda* is a parameter in the domain in which this translation relationship is defined, two versions of lambda are automatically initialized: *old_lambda* and *new_lambda*. The last line in the script gives the scaled area.

```
translation -scale area {
        #
        # $value is the known data point.
        #
        # $lambda is a parameter in this domain, so
        # $old_lambda and $new_lambda automatically get set up.
        #
        # area scales with the ratio of the lambda's
        # s = old_lambda/new_lambda
        # new_area = old_area / s^^2
        #
        set one_over_s [expr $new_lambda/$old_lambda]
        expr $value*$one_over_s*$one_over_s
}
```

In the second example, the variable *value* is also set up automatically, this time the delay value that needs to be scaled. The script defines an empirical scaling relationship that uses a data set consisting of (voltage,scaling factor) tuples. In this case, the script assumes that the delay to be scaled has been estimated given a 5.0 V supply voltage (note the last tuple (5.0,1.0) indicates a scaling factor of 1 at 5 Volts). Thus, a scaling factor can be found from the data set using the utility routine "polate," and the new delay value is found by simply multiplying the original delay by the scaling factor. Again, the last line in the script gives the scaled delay.

```
translation -polate delay {
        #
        # "voltage" is a parameter in this domain, so
        # $new_voltage automatically get set up.
        #
        #             x1  y1     x2  y2   ...
        set dataset {{1.5 6.9} {2.0 3.5} {3.0 1.8} {4.0 1.2} {5.0 1.0}}
        set scaling_factor [polate $new_voltage $dataset]
        expr $value*$scaling_factor
}
```

Stimulus Templates - a list of reusable templates for rapid test bench generation.

A stimulus template is a procedure that generates a generic list of tokens (typically {time,value} pairs) which can be used to (semi-)automatically create test benches. Some design and analysis tools (especially simulators) require not just design files but also a test bench or test suite. The problem is that each tool has its own unique format for describing test benches, which slows the verification process. A test bench defines values for each input in a design, usually with some notion of time. It is often desirable to be able to use the same test bench at various levels of abstractions throughout the design process. Therefore, a generic format that can capture the values to apply to each input is required. Prior to invoking a simulation tool, a test bench can be created on the fly in the required format, based on the generic format. This is a two step approach: first a generic list is created based on the stimulus templates, then it is translated to the specific format required. The translation process is defined on a "per tool" basis as part of a tool's encapsulation (as a "preparation script"). Stimulus templates, in addition to the script that creates the generic list, have two options: a "match" option that specifies a regular expression, i.e., a pattern for how to guess if an input name is likely to use this template (see examples below), and zero or more "parameter" options giving a parameter name and a default value. If a parameter name is the same as one of the domain parameters, then the domain parameter's value will be used.

Syntax: numgen name [-match regexp] [-parameter_name default_value] ... script

Example: numgen ground -match {^gnd.*$} {

    lappend timevaluepairs [list 0 0]

}

This example defines a very simple template for a ground (0 Volts) signal. It has a "match" option which specifies that a signal name that begins with the letters 'gnd' and has any number of characters (zero or more) after that are likely to use this template. A designer will always have a chance to override this guess, but it provides a good starting point. The script creates a list {0 0} ({time value}) which means this signal should have the value 0 at time 0. Any other interpretation of these values is done in the translation scripts defined in a tool encapsulations.

Example:

```
numgen reset -match {^r[e]?s[e]?t.*} -delay 0 -activevalue 1 -duration 1 {
      set initialvalue [expr {$activevalue ? "0" : "1" }]
      set time [expr 0*$timescale]
      lappend timevaluepairs [list $time $initialvalue]
      set time [expr $delay*$timescale]
      # Make first transition
      lappend timevaluepairs [list $time $activevalue]
```

```
            # Stay active for the duration.
            set time [expr $time+$duration*$timescale]
            # lappend timevaluepairs [list $time $activevalue]
            # Make second transition
            lappend timevaluepairs [list $time $initialvalue]
    }
```

This example defines a template for a reset signal. Reset signals can often be recognized by their names, beginning with "reset" or "rst." This template has three parameters: delay, activevalue, and duration. The delay indicates how long to wait before asserting the signal. The activevalue indicates whether this signal should be asserted high or low. The duration indicates how long the signal should be asserted. See Figure 5-2 below.

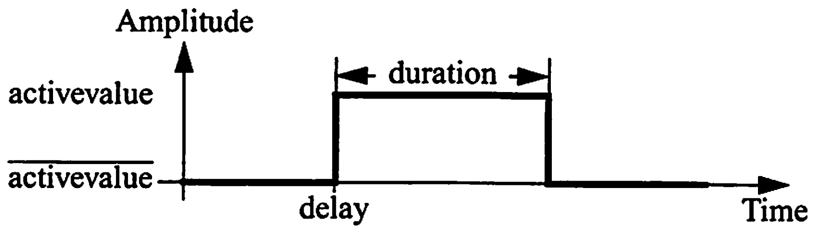


Figure 5-2. Reset Signal.

Example: The most generic template uses values from a file and turns them into a list. The following example reads one real value (floating point) from each line of a given file (see Figure 5-3).

```
numgen filereals -match {^.*$} -file data -sampledelay 0 -sampleperiod 1 {
        # Wait for sampledelay, then read the samples
        # at sampleperiod intervals.
        set fp [open $file r]
        set time [expr $sampledelay*$timescale]
        set max_time [expr $simtime*$timescale]
        while {($time < $max_time) && ([gets $fp line] != -1)} {
                lappend timevaluepairs [list $time $line]
                set time [expr $time+$sampleperiod*$timescale]
        }
        close $fp
}
```

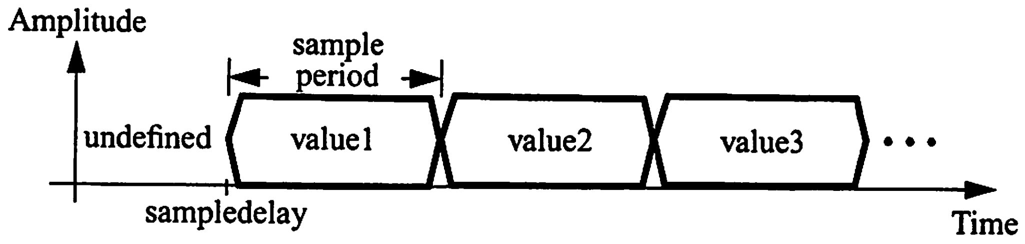


Figure 5-3. File-of-reals Signal.

5.1.4. Domain Operations

As mentioned in the previous section, attributes and operations carry the information content of a domain. Attributes contain the static elements of the information, and operations contain the dynamic information, namely the methods that define run-time behavior. There are two aspects of operations: their implementation, and their presentation to the designer. These two aspects are explained below.

Methods - one or more implementation(s) of an operation.

An operation can have several different implementations, each operating on different data. For example, an "area" operation for estimating the area of a chip design can have one implementation for extracting the area of a layout, and another for estimating the area of a structural netlist. A method is defined in a script written in the extended TCL language (Tool Command Language) described in Appendix A and B. Scripts consist of a descriptive name, the name of the data resource type on which the script can operate (if any), and an executable body. Scripts can call tools (using the tool encapsulations described above), or other scripts. There can be several scripts with the same name, in cases when different techniques can be used to perform a given task.

Syntax: meth name filetype script

Example: meth area netlist-file {

    # Tcl script that defines how to estimate the area of a netlist.

    ...

}

Requests - a list of operations and how they should be presented to a designer.

The request list defines which operations are suitable for use directly by designers, and provides a descriptive label for each operation. The label is presented to a user (through a user interface, see Section 5.6.) and should be as descriptive or intuitive as possible. The operations that don't appear in the request list are typically only called from other operations.

Syntax: request_type label [-group name] [-method method -filetypes list_of_types]

where request_type can be either view, analysis, optimization, or action.

Example: view "Chip Area" -group "Cost" -method area -filetypes {netlist-file lay-out-file}

For some examples of domains, the reader is referred to Appendix D.

5.2. Contexts

The purpose of a context is to bring together all of the knowledge that is relevant to a given design. Thus, a context can be empty if no relevant knowledge has been identified or is available, or it can contain one or more domains. It is necessary to establish a set of rules for how a context can be composed, to avoid contexts that have internal conflicts. As an example of a context with an internal conflict, consider a context consisting of two domains, one which says "fabricate the design in plastic," the other which says "fabricate the design in aluminum." These two domains constrain the same area of expertise, but their constraints conflict. A set of five rules for creating and modifying contexts are presented below.

5.2.1. Rules for Context Creation

A set of rules is needed to avoid joining incompatible domains to a context. The rules presented below determine if a given domain can be added to a context, based on the domains that are already members of the context.

Rule 1. If the context already contains the domain to be added, it is not added again.

Rule 2. If the context contains a domain that is derived from the domain to be added, the new domain will not be added. This is done to preserve the more specific domains in a context. If a designer wishes to defeat this rule, i.e., to add a less specific domain than what is already contained in the context, then the more specific domain has to be removed before the less specific domain can be added.

Rule 3. If the context contains a domain that belongs to the same family of domains (i.e., they are derived from the same root domain) then the new domain replaces its family member. This is done to avoid having conflicting domains in the same context, such as a 1.2 μm technology domain and a 0.6 μm technology domain.

Rule 4. When a domain is added to a context, all the associated domains that are listed in the domain's specification are also added, subject to rules 1 and 2.

Rule 5. A designer is always allowed to remove a domain from the context, since this action can not create a conflict. However, if the removal causes information which is used in other domains to be missing from the context, an error would occur. Therefore, after domains have been removed from a context, but before new requests are handled, the environment has to ensure that the context complies with rule 4 above.

In designs consisting of a hierarchy of components, each design component is allowed to have its own unique context.

5.2.2. Rules for Merging Domain Contents

A set of rules is needed to govern the process of merging domain contents. The process is very simple when there are no conflicts between attribute or operation names. In those cases all attributes and operations are visible in the context. However, if there are conflicts there has to be a set of rules to help resolve them. The rules below assume that a context initially is empty; each inherited domain is visited in turn, and each attribute and operation from that domain is considered one by one. These rules then determine whether or not an attribute or a method will be visible in (i.e., will be added to) the context.

Rule 1. If the context already has an attribute with the same name as the one currently considered, the current attribute is not added again. Due to the concept of orthogonality, this occurs infrequently.

Rule 2. If the context already has a method with the same name and with the same arguments as the one currently considered, the current method is not added again. However, if method names are the same, but argument lists differ (i.e., they operate on data of different types), the current method is added.

5.3. Design Objects

A design object is an internal representation of a specific design. In object-oriented terminology, a design object is an instantiation of the context class. However, the instantiated context is allowed to change its inheritance dynamically, which means the information resources (domains) that are available to a design object can change throughout a design process.

The main purpose of a design object is to keep track of all given design specifications, as well as a list of design files and the current values of parameters and constraints. Design objects also generate and maintain history records for a design, including tool invocations and file derivation relationships. In addition, a design object provides the environment for command interpretation and execution.

The next three subsections describe how commands are interpreted and executed by a design object, and how a design history is automatically generated. The last subsection explains how design objects are used to represent hierarchical designs.

5.3.1. Command Interpretation

A design object only has access to the knowledge contained in its context, so only the methods that are defined in the inherited domains are visible to the object. Therefore, when a designer gives a command, the design object must determine the meaning of the command, based on the available knowledge. Commands can be overloaded, which means that within one context there may be several different interpretations of the same command, and as a context evolves, the meaning of a command can change.

The command interpretation process works as follows.

1. A designer issues a request for information, typically in the form of a word, e.g., "power".

2. The context tries to match the request with the names of the available methods. If there are no methods that match the request, the designer is notified that this request cannot be handled.

3. All the methods that match the request are inspected. If a method requires as its input a file of a certain type, it is determined if such a file is available. If there are no files of the right type, the given method is rejected.

4. For the remaining methods, the lists of files of the proper types are inspected. In each list, all files that are derived from other files are removed. (A file is derived if it has been created as the result of running a tool with another file as input. This information

is maintained in a history record). See example below.

5. If there is only one method left, and the method has only one file in its list of files, or it doesn't require a file, it is automatically executed. If there is more than one, the designer is asked to decide which one to use, and the chosen method is executed.

Example This example illustrates steps 3-5 above. All the methods in the table below are assumed to match the given request (step 2). Step 3 corresponds to column 3, step 4 corresponds to column 4 and step 5 is explained below.

All methods matching the request:			
Name	Filetype	Available Files	Non-derived Files
meth1	type1	none	
meth2	type2	file1, file2	file1, file2
meth3	type3	file3, file4	file3

The file derivation relationships assumed in this example are shown below. An arrow indicates that the file to the left of the arrow was the product of a tool execution that used the file to the right of the arrow as one of its inputs.

$$\begin{tikzcd}
 & \text{file1} \arrow[r] & \text{file3} \arrow[r] & \text{file4} \\
\text{file0} \arrow[ur] \arrow[dr] & & & \\
 & \text{file2} & &
\end{tikzcd}$$

In step five, it is determined that there are three possible interpretations of the request. Therefore, the designer is asked to choose between the three alternatives: (meth2,file1), (meth2,file2), or (meth3,file3).

5.3.2. Execution Environment

The process of interpreting a request determines which method to use, and which file, if one is required, to use as the input to the method. The method is defined in an executable script. Before the script is executed, the design object ensures that all parameters defined in the context, with their current values, are visible to the script. In addition, a number of variables are initialized, which can be used by scripts to facilitate the proper handling of results, units, and details.

RESULT - this variable can be set by the script to any string that will be returned to the caller. The caller can be the designer or another script.

UNITS - this variable can be set to indicate the units of a result, e.g., "mm^^2" (mm^2). The variable, if set, is used to facilitate the automatic conversion of units, if necessary. For example, if one script returns a result in units of "mm^^2" but the calling script is dealing with "um^^2" (μm^2), then the environment can automatically perform the conversion (provided a conversion can be identified, either using standard prefix analysis, or using translation scripts defined in a member domain).

DETAILS - this variable can be set to contain a list of details about the result. This can include measures of certainty of a result (e.g., +- 10%), or how the result was calculated, etc. Since this is a list, any number of details can be added.

The design object automatically records a trace of the methodologies and tools that are called during the execution of a script, and the trace can be inspected after-the-fact to determine how a result was obtained. This can be very useful, especially when several methods are called from the initial script or when deep nesting occurs.

If a script (a method) calls other methods, it can do so in two different ways. It can either specify a (method,file) pair, in which case resolution is not necessary. Alternately, it can specify just the method, in which case it will be interpreted in the same way as a command given by a designer.

Example These examples show how methods can use the three special variables.

```
meth area layout-file {
	# Declare the preferred units (sizeUnits is a domain parameter)
	set UNITS "${sizeUnits}^^2"
	# Extract area from layout file.
	...
	set layout_area ...
	...
	# If possible, extract area of various components
	# (e.g., active area, routing area, I/O pad area, etc.)
	...
	lappend DETAIL [text "Area Breakdown" Active: $active, Routing: $routing]
	...
	# Set the RESULT variable:
	set RESULT [data area $layout_area $UNITS float]
}

meth area hierarchy {
	# Declare the preferred units (sizeUnits is a domain parameter)
	set UNITS "${sizeUnits}^^2"
	# Calculate total area by summing up the area of each subcomponent.
	# Use the built-in "sum" function which passes the "area" request
	# to each subcomponent of this design object, and also creates a DETAIL
	# that explains each component of the sum.
	set RESULT [sum area $UNITS [subcells] DETAIL]
}
```

5.3.3. History

A design object automatically generates two kinds of history records. The first is an execution trace which records sequences of which scripts have been called and which tools have been executed. As mentioned above, traces are generated for the benefit of designers, to determine how results were generated. The second type of history record maintains a detailed list of tool invocations, including the exact tool arguments and invocation string used. This history is generated for use by the design environment to avoid re-executing tools that have already been executed.

Example This example shows the structure of the history records. For each tool that has been invoked there is a list of unique identifiers for each invocation:

```
HISTORY(toolname,indices) {index1 index2 ...}
```

For each identifier there are six entries that capture the specifics of a tool invocation: the input arguments, the outputs produced, the environment that was established before the tool executed, the actual invocation string used, the script used to determine success or failure, and finally the result (which can either be "SUCCESS," "FAILURE," or "PENDING" for a background job in progress). [Note: the two lines that begin with a '#' are comments for the benefit of the reader]

```
HISTORY(toolname,index1,inputs) {
   #variable        filename  filetype     fileobject  required/optional?
   {silFile         ncd.sil   silage-file  FILE0       REQ}
   #variable        value     type         n/a         required/optional?
   {asciiFlag       -a        string       {}          OPT}
   {hyperFlag       -H        string       {}          OPT}
   {delayLineSample 16        string       {}          OPT}
   {delayLineLoop   1         string       {}          OPT}
}
HISTORY(toolname,index1,outputs) {
   {aflFile ncd.afl hyper-flowgraph-file FILE12}
}
HISTORY(toolname,index1,environment) {
   setenv hyper ~hyper/hyper-new ; source $hyper/hyperscript
}
HISTORY(toolname,index1,execution) {
   mm -a -c -H -D 16 -E 1 ncd
}
HISTORY(toolname,index1,status) {
    if {$status == 0} {
        return "SUCCESS"
    } else {
        return "FAILURE:\nErrors occurred while running $name:\n$result"
    }
}
HISTORY(toolname,index1,result) {
    SUCCESS
}
```

5.3.4. Hierarchy

Design objects represent designs, or components of a hierarchical design. Therefore, objects can optionally contain a component list, which identifies other design objects that are instances in the current object. This representation is not intended to replace design specifications given in design files or databases. It only provides a minimal representation which the design environment can manipulate. Whenever a methodology script is intended to work on the design object hierarchy, as opposed to a design file, the word "hierarchy" is used instead of the name of a file type in the domain specification. Such a script is considered enabled (i.e., its preconditions are met) when an object has one or more elements listed in its component list.

5.3.5. File Type Recognition

The purpose of file type recognition is to be able to determine the type of a file, such that a design object can determine how the file can be used. File type recognition is in principle a generic utility, but is important in the information-centric environment because it enables the environment to automatically match a tool or a method with a given design file. The type recognizer uses the file type definitions, specified in domains, to recognize the type of a file. Two approaches are used, namely name pattern matching and content matching. In name pattern matching the name of a file has to match a pattern. For example, the pattern for C files is *.c, and a file by the name file.c will match this pattern. A "*" matches any sequence of characters in a string, and a "?" matches any single character in a string. For content matching, an offset (integer) and a string (character string) must be given. The recognizer will read the first offset+length_of(string) bytes from a file, throw away the first offset bytes, and compare the result with the given string. Since the content matching is considerably slower, the name matching is always performed first.

The file type recognizer has a feature not normally found in recognizers: it can propose a name when a designer wishes to create a new file. This is done by replacing the first "*" by the name of the design object, and all "?" by an arbitrary character (the character "0" was chosen). The resulting name can be used as is, or the designer can propose another name.

5.4. Search Engine

The context is very important in an information-centric design environment, because it forms the basis for the aid that can be given to a designer. It is clear that a designer must be given full control over which domains are members of a context. However, in the presence of a large number of domains, it is not always clear which domains are appropriate to a given design, or to the given stage that a design is in.

Therefore, it is necessary to provide a search mechanism that can operate within the specification driven model of an information-centric environment. The goal of the search mechanism is to locate relevant domains, and propose a context that best matches given design specifications.

The following four sections define the problem which the search mechanism is intended to solve, outline the supported syntax, describe the search strategy, and explain how the designer can interact with the search results. Finally, a special case is considered, namely when a design specification is accompanied by a command.

5.4.1. The Search Problem

The search problem can be defined as follows:

Given a list of words representing the design specifications, find all domains that contain relevant knowledge. Order these domains in terms of their relevancy, and create one or more contexts. Order the contexts and propose one or more to the designer.

5.4.2. Syntax for Specification Driven Search

The search strategy must be able to exploit the specifications that a designer gives in the specification driven design model. A specification consists of one or more keywords. Each keyword is can have the following syntax:

```
keyword:
    word
    word=value
```

word:=value

Each of these cases trigger a different search approach and is described below:

word

When a word (a character string) is specified without an accompanying value the search strategy will look for exact and partial matches in all categories of domain knowledge (e.g., identifiers, parameters, constraints, method names, etc.).

word=value

When a word and a value are specified together with an equal sign, the search strategy will only search domain identifiers and parameters. If an identifier or parameter name matches the given word, the given value is matched with the list of identifiers or the value of the parameter.

word:=value

When a word and a value are specified together with a colon-equal sign combination, the search strategy will only search domain parameters. If a parameter name matches the given word, it is counted as if a regular word was found. The value is saved so that if the domain containing the parameter is used in a context, the parameter's value will be set to the given value in that context.

5.4.3. Search Strategy

The search strategy outlined in this section has been implemented in a search engine which is an integral part of the design environment. Before describing the search strategy and the approach to ordering the results, a number of assumptions will be delineated.

Assumptions - the search engine makes three assumptions:

1. A list of one or more specifications are given in the syntax described above.
2. An ordered list of all available domains can be obtained from the design environment. For domains that share a common ancestor, more general domains are listed before more specific domains.
3. The relevance of a domain to the design specifications can be estimated by the number of exact and partial matches found in the domain.

Domain Search - during the search process the engine traverses all domains looking for exact and partial matches for each of the given search words. When all domains have been searched, four scores are assigned to each domain. The first two are the number of exact and partial matches found in the domain. The last two also represents exact and partial matches, but they are summations of matches over all the domain's associated domains (for an example, see the end of this section).

Domain Ordering - all domains are ranked according to the 4 scores of exact and partial matches. The total number of exact matches in a domain and its associated domains is used to create an initial ranking. If domains are tied, the total number of partial matches in a domain and its associated domains is used to resolve the tie. To resolve further ties the exact and partial matches in a given domain are considered.

Context Composition - a new context is gradually assembled starting from a clean slate. The highest ranking domain is chosen as the first to be added. If that domain and its associated domains contain matches for all the search words, the context is considered complete. Otherwise, the ranked list of domains is consulted to find a domain which matches search words not yet matched by the context, and which is not in conflict with the domains in the context. If found, this domain is added along with all of its associated domains, subject to the context creation rules, and it is verified whether the resulting context contains matches for all search words. This process is continued until all search words that can be matched are covered by the context. If there are search words for which no matches are found in any domains, the designer is notified.

Alternate contexts can be composed in a similar fashion, under these considerations:

- Every time a domain is chosen from the ranking list, check if there are other domains in the list that have the same qualifications (rank) as the chosen domain. If so, continue developing the context with the chosen domain, but also create a new context that replicates the first, except for the chosen domain.
- When the context(s) have been assembled that used the highest ranking domain as the first to add, create new context(s) starting with the second highest ranking domain.

The first context that is created is the search engine's "best bet," and all others are the runners up. Designers are free to choose any of the proposed contexts, and to modify the chosen one.

User Interaction - there are three different scenarios of how the search engine presents its results. The scenario chosen depends on the number of contexts that were developed.

- No contexts: this implies that the given specifications did not help identify any relevant domains. In this case the designer is notified and asked if another search is desired.
- One context: this implies that only one context appears to make sense based on the search specifications. In this case, the context is proposed as a "take it or leave it" solution.
- Many contexts: this implies that several contexts are feasible, and some contexts may be equally relevant to a design. In this case, the ordered list of contexts are presented, and it is left up to the designer to choose which context appears to better reflect the intentions of the specifications.

Extensions - the search strategy does not address the issue of finding domains that are not already known. Two possible extensions to the strategy involve searching local file systems for domains created by other designers, and using established Web search approaches to locate domains available on the World Wide Web. A third extension is to contact other design environments and automatically exchange domains. These extensions are beyond the scope of this work, but will be addressed in chapter 8 "Future

Work."

Example This example shows how scores are assigned to domains during the search process. Also, the context that will be proposed is described. As mentioned above, four scores are assigned to each domain. They are:

1. the number of exact matches found in the domain
2. the number of partial matches found in the domain
3. the number of exact matches found in the domain AND in all associated domains
4. the number of partial matches found in the domain AND in all associated domains

Figure 5-4 shows four domains. The generic cell library domain declares an association with the generic technology domain. The four scores that are assigned to each domain are shown. Based on the scores for all exact matches, the domains are ranked as follows (two ties):

```
Rank 1 - Domain Adder, Domain 0.6 um
Rank 2 - Domain Cellib, Domain Technology
```

The scores for all partial matches are used to resolve the ties between the adder and the 0.6 um domains. The following ranking results:

```
Rank 1 - Domain Adder
Rank 2 - Domain 0.6 um
Rank 3 - Domain Cellib, Domain Technology
```

Further resolution of ties is attempted using the exact and partial match scores, but in this case, no further resolution can be made. The context creation begins by creating a context that inherits the adder domain. The adder domain alone has matches (exact or par-

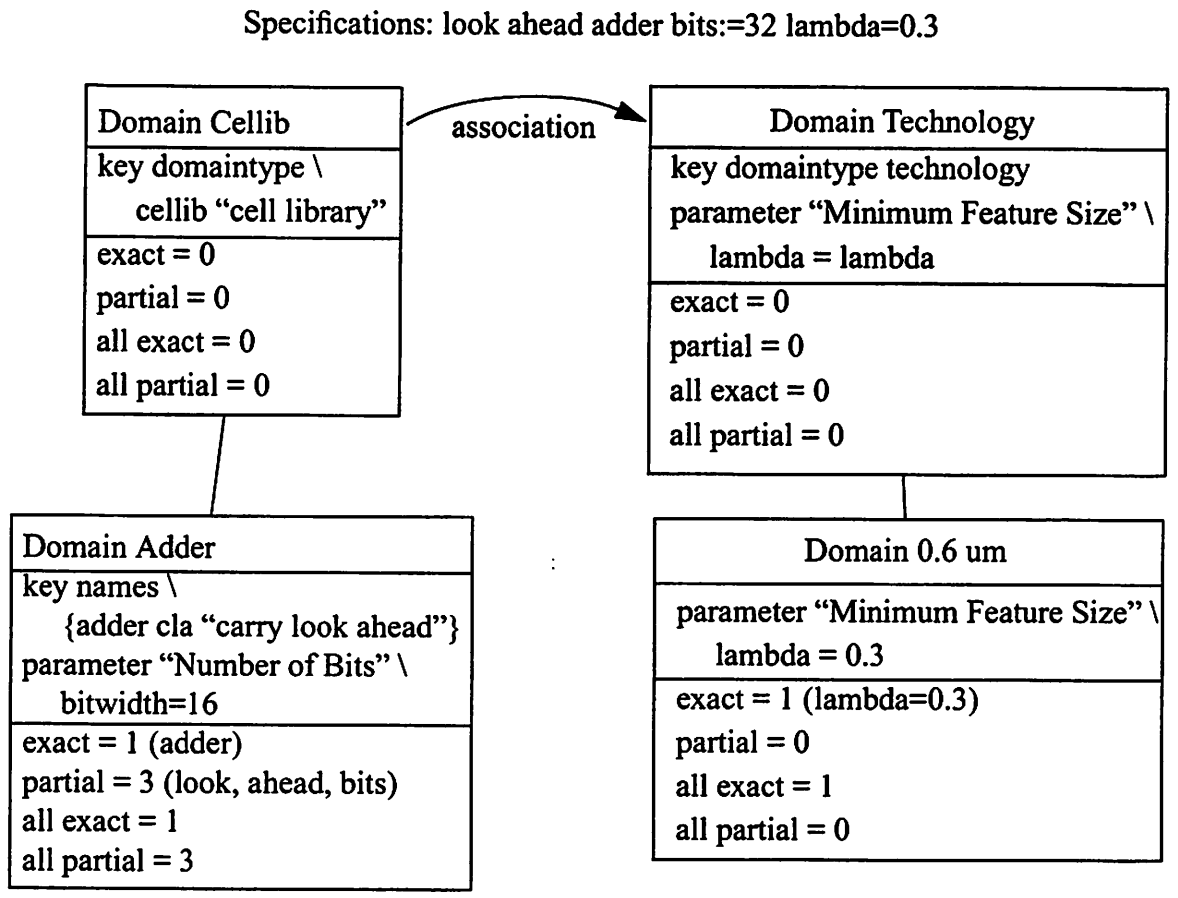

Specifications: look ahead adder bits:=32 lambda=0.3
Domain Cellib
key domaintype \ cellib "cell library"
exact = 0
partial = 0
all exact = 0
all partial = 0
association
Domain Technology
key domaintype technology
parameter "Minimum Feature Size" \ lambda = lambda
exact = 0
partial = 0
all exact = 0
all partial = 0
Domain Adder
key names \ {adder cla "carry look ahead"}
parameter "Number of Bits" \ bitwidth=16
exact = 1 (adder)
partial = 3 (look, ahead, bits)
all exact = 1
all partial = 3
Domain 0.6 um
parameter "Minimum Feature Size" \ lambda = 0.3
exact = 1 (lambda=0.3)
partial = 0
all exact = 1
all partial = 0

Figure 5-4. Scores assigned to domains during the search process.

tial) for (adder, look, ahead, bits), but not for lambda=0.3. Therefore, the next domain in the ranked list which has a match for lambda=0.3 is added to the context inheritance, resulting in a context that has matches for all the search terms. This context is proposed to the designer.

Other Comments - As the above example hints at, it is quite possible for specifications to be meaningful for a designer but fail to select the desired domains. The specification "bits" is meaningful for most designers, but it failed to match the "bitwidth" parameter variable name. Due to good luck the parameter label "Number of Bits" produced a partial match, which was sufficient to ensure that the right domain was selected in this case. It is therefore very important that domain developers choose the most intuitive, common words, and provide plenty of alternatives, as illustrated by the values given for the key "names": {adder cla "carry look ahead"}.

5.4.4. Single Request Strategy

To facilitate quick feedback, the search engine can be used in a slightly different way. The designer can include a command in the specifications, with the intent that the search engine should find what it believes to be the best context, and then execute the given command within that context. This feature enables designers to make single requests for information such as:

- what is the delay of a multiplier cell?

  Specifications: delay multiplier (or: delay *)
- what is the delay of a multiplier cell in a 0.6 μm technology?

  Specifications: delay multiplier lambda=0.6
- what is the computational complexity of a 1024 point FFT?

  Specifications: complexity FFT points:=1024
- what is the power consumption of an LCD display?

  Specifications: power LCD

The strategy used to respond to these requests is simple: The search engine proposes a list of contexts, the highest ranking is chosen, and the command is interpreted and executed in that context. If there are many ways to interpret the given command, the designer is consulted.

Another variation of this feature is illustrated with the following request:

- what is the area of all adder cells?

  Specifications: area -all adder (or: area -all +)

In this case, one context is chosen to represent each domain which matched the search word adder (or +). For each of the representative contexts, the command is evaluated, and the results are presented to the designer. This feature has not been implemented in the current prototype, but may be considered for inclusion in future versions. The syntax of the specification language would have to be expanded to declare "-all" a reserved keyword, and the search engine would have to recognize this word and provide the desired functionality.

5.5. Resource Management

This work is not trying to replicate the resource management from traditional CAD frameworks. It would be beyond the scope of this work to do so, and previous work in this area has yielded good strategies and implementations for managing the resources available to a CAD framework. However, an information-centric design environment has to handle some of the management tasks that are related to data, tools, and design flows. This section describes the necessary capabilities.

The resource managers that are needed fall into three categories of data, tool, or design flow managers. Their purpose is to exploit the information captured in domains to provide seamless integration. For the data manager, it amounts to distinguishing between design data of different types. For the tool manager, it amounts to automating tool invocations by correctly composing invocation strings. For the design flow manager, it amounts to providing an environment in which design flows can be executed and where results and errors can be properly handled. The next three subsections explain each of the three managers in more detail.

5.5.1. Data Management

The main function of data management is to manipulate design files. An object oriented internal representation, called a file object, is used. A file object contains information such as the name of a file, the directory in which it resides, the type of the file, and which tool, if any, produced the file.

When a tool is about to be executed, it is determined if any files are going to be produced. This can be determined from the tool encapsulations specified in domains. If an existing file will be overwritten by the tool, the data management will move the existing file out of the way by renaming it. A table that maps actual file names to file objects is maintained, and files that have previously been renamed will automatically be restored next time they are used.

5.5.2. Tool Management

The main goal of tool management is to properly execute design tools, and to report whether they failed or succeeded. The step by step tasks of the tool management are shown below.

Gather Inputs - the inputs to a tool include input files and the parameters that are used to specify different tool options. When a tool is called, the inputs are gathered from one of three sources. If a value is given in the tool invocation, that value is used. Otherwise, if an input parameter has the same name as one of the parameters in a domain, the current value of the parameter is used. Otherwise, the default value specified in a tool encapsulations is used.

Predict Outputs - based on the inputs, use the knowledge in the tool encapsulation to predict the names of the files that are expected to be produced by the tool.

Create Run Time Files - for tools that require files in addition to design files, e.g., simulators need command files. This may require interaction with the designer.

Create the proper execution environment - tool encapsulation specifies the necessary environment variables, paths, etc. Tool management takes care of creating the proper environment.

Execute tool - start the execution of the tool, either as a foreground or background process, keep track of process ID numbers (to enable cleaning up long jobs if the design environment is killed).

Determine status - determine whether the tool's execution was a success or a failure. Most tools exit with a status code, and the encapsulation specifies how to interpret the code for the given tool.

In addition to the above steps, the tool management informs the data management about which files will be created by a tool, in order to enable the data management to create backups if necessary. Also, a history is kept with all the inputs, outputs, etc., which helps determine whether a tool has already been invoked and does not need to be invoked again.

5.5.3. Design Flow Management

The design flow management is the same as the command execution environment discussed in Section 5.3.3.

5.6. User Interface Issues

As with all interactive computer software, there has to be a way for the user to interact with the design environment. The interface has to allow designers to conveniently access the available functionality, and to gather all of the information that the environment can provide. The following sections describe the model chosen for the environment, and discusses the pros and cons of this approach.

5.6.1. Client-Server Model

Our design environment uses a client-server model, in that it acts as a server that responds to requests made by a designer through a graphical user interface (GUI) client,

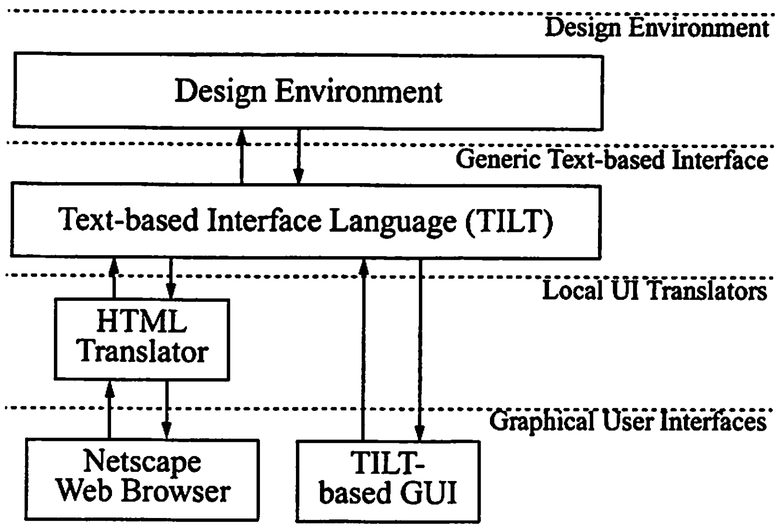


Figure 5-5. Client-Server Model Uses External User Interfaces.

which can be a dedicated GUI or a design tool. In this model, the user interface is a separate, or disconnected, entity that is not integrated into the environment. Since the user interfaces are not built-in, we developed a generic text-based interface language for the environment (called "TILT", see Section 5.6.3. below) which can easily be translated to the requirements of specific interfaces. Currently we have an HTML translator as shown in Figure 5-5.

5.6.2. Benefits of the Client-Server Model

The benefit of the client-server model is that the user interface can be customized for many different applications to suit the specific needs of a group of designers. The model also readily allows the environment to be encapsulated by another design tool, provided that the tool is capable of dynamically changing its command menus.

An even bigger benefit of the client-server model is that a designer can access the environment from anywhere in the networked world. The communication channel between the client and the server uses TCP/IP, the same technology as is used in the World Wide Web. Thus, designers can access the environment from any terminal ranging from a portable, wireless terminal such as the InfoPad, to a stationary desktop workstation.

5.6.3. A Request and Feedback Language

The interactions between a designer and an information-centric design server are depicted in Figure 5-6. A designer makes a request, and the server responds by giving feedback.

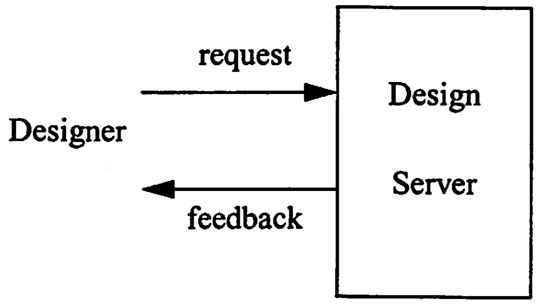


Figure 5-6. A Designer Makes Requests, the Server Provides Feedback.

Since designers can use clients that require different formats, the design server provides a generic request and feedback language that can easily be translated to different formats. A generic text-based interface language, called “TILT,” was developed for the design server. This language will be described in the “Requests” and “Feedback” sections below, and in greater detail in Appendix G.

REQUESTS Requests can either be directed at a specific design object or at the design server to control resources that are global to the server.

1. Requests to a design object.

   Requests to a design object take the form

   designObject request [arg1] [arg2] ...

   The word "designObject" should be replaced with the name of a specific design object, and it ensures that the following request is directed to the right design object. The request can be one of several different keywords (e.g., menu, context, specifications, subcells, del (for delete), save, trace, files, run, set/getParameter, etc.) but the most useful is the "menu" request. The menu request prompts a design object for all currently available requests, which includes the "permanent" requests that are always available (e.g., context, save, del (for delete), etc.) and the dynamically enabled commands that are defined in domains and are only available when they are able to execute. A designer normally gives the "menu" request to a design object first, then selects one of the available commands through the interface client. Thus, the designer doesn't need to worry about the proper request key-word to use, or the right arguments to pass along, it is all contained in the feedback to the "menu" request.

2. Requests to the design server.

   Requests to the design server are used to access global resources. There are six requests available (the arguments shown in brackets are optional):

   openDesign [design] - Access another existing design object.

   newDesign [design] - Create a new design object.

globalOptions - Access the server's global options.

The global options include defaults for search paths, whether or not to dispatch jobs to remote hosts, whether constraints should be checked or not, etc.

menuDomains - Show the available domains.

computerMenu - Computer management.

Show the available remote hosts and their work load. The workload of remote hosts is automatically monitored when the global option to dispatch jobs to remote hosts is set.

shutdown - Shut the design server down.

This request causes the design server to exit after killing all tools that may still be running.

FEEDBACK The feedback language needs to be sufficiently rich that it enables a GUI to display complex messages properly, yet it needs to be sufficiently simple to allow for easy translation to specific interface formats. Design server feedback contains "elements" of different types. Each of these types have to be distinguishable in the feedback language. The feedback types are either for conveying information to the designer, or for allowing the designer to enter information to be returned to the server (e.g., choosing between several options). The types are *text, data, command, menu, options, table, and text-entry.* Appendix G specifies the exact format of each of these types, but some of them deserve a little more explanation at this point. The menu type is not strictly necessary, but it offers a convenient and familiar way to present a list of commands. The options type is used to convey available options, and lets the designer choose between them. There are two sub-types including select-one-of-many and select-zero-or-more options. The table type is used to structure feedback in a two dimensional row and column layout. The text-entry type enables a designer to enter one or more lines of text (e.g., when giving the name of a design object to create).

The feedback is structured as a set of nested lists. Lists are textually represented by **{ element1 element2 ... }**, and nested lists form one or more elements in other lists, such as **{ element1 {element2_1 element2_2 ... } element3 }**. Feedback is always returned to a client with the first element of the top-level list set to the keyword "document." The document list can have any of the other types nested in it.

Example The following example is self-explanatory:

```
{document {text "Hello World"} {text "How are you?"}}
```

The next example shows how commands can be nested in text, which in turn can be nested in a document:

```
{document {text "Command Selection" {command designObject request1 arg1}
{command designObject request2 arg2 arg3}}
```

5.6.4. Challenges for the Client-Server Model

Supporting a disconnected user interface poses a number of challenges. First of all, there is the issue of security, since the design environment can be accessed from anywhere in the World Wide Web, user identification is an important issue. Since the environment has full access to a user's file system (i.e., can read and write files), much damage can result if an intruder obtains access. We have implemented an identification strategy that requires a user to give a name and a password. However, when web browsers are used to access the environment, it would be tedious to have to give a name and password for each interaction. We have augmented the HTML translator to support the "cookies" that are used by popular web browsers, such as Netscape Navigator or Microsoft Explorer. In web terminology, a cookie is a token which the server (in this case the design environment) passes to the web browser when the user has been properly identified, and which the web browser uses as a special password for successive accesses.

Another important challenge arises from the disconnected UI model, since both the user interface and the design environment carry information related to a certain state of a design. Since the environment has no control over the disconnected UI, it is possible that the UI becomes outdated with respect to the environment. The state of a design is contained in a design object. The state information in the UI is contained in a list of the available commands for the given object. Therefore, the design environment must be able to determine if a command originated from an outdated list, and notify the designer to update the user interface when necessary. This is accomplished by giving each context an identifying number which is annotated on each list of available commands. If a request is received with a different identifying number than the current context, the UI needs to be updated.

The interface also has to provide useful information about what the design environment is doing at run-time. The environment offers two different means for that: execution traces and run-time information. Execution traces are automatically generated whenever the environment executes scripts or tools. These traces can be used as an after-the-fact way of examining what the environment did. Run-time information (e.g., decisions that are made, tools that are invoked, etc.) is a more immediate indicator of what the design environment is doing. In the disconnected UI model, it is necessary to be able to redirect run-time information if requested by the UI. Normally, run-time info is directed to the terminal where the design environment is started, but upon request it can be sent somewhere else, for example, to a Web browser. If the browser terminates the connection, the run-time info is directed back to the original terminal.

5.7. Summary

Domains are object classes that encapsulate design expertise, and they are organized in families of object-oriented classes. A more refined (more specific) domain is derived from a less specific parent domain and it inherits the expertise in the parent domain, but can override any of the expertise or add to it as necessary. Contexts are object classes that have multiple domain inheritance, and the inheritance can change dynamically under user control. Design objects are instantiations of contexts, and provide a number of utilities to control the dynamic inheritance, keep track of design files and history, etc. Design objects are the focal point for designers, since most requests or commands are posed to a design object. The next chapter describes a prototype information-centric environment.

CHAPTER 6

THE DESIGN SERVER

A prototype design environment, called the Design Server, has been developed as a part of this project, and it embodies the information-centric design model. I implemented the prototype of the system in a version of the Tool Command Language (TCL) [Ousterhout94] that supports object oriented programming (incr-Tcl) [McLennan93], TCP/IP communications (Tcl-DP), and interactive program control (Expect) [Libes95]. This version of TCL is presented in detail in Appendix A.

This chapter presents a flow description of typical design sessions using the Design Server. Chapter 7 presents some specific design examples using the Server. For additional information, Appendix H discusses some practical considerations about how to start the Server and interact with it through external interfaces.

6.1. Design Flow Description

A typical design process using the Design Server is illustrated in Figure 6-1. The ini-

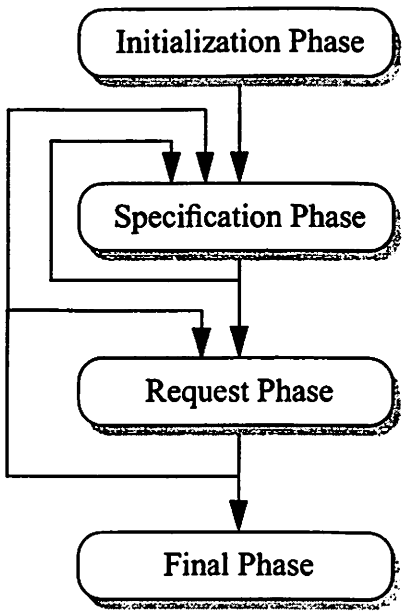


Figure 6-1. Design Flow Diagram.

tialization phase is described in Section 6.1.1. The specification phase is described in Section 6.1.2. and the request phase in 6.1.3. The final phase is described in 6.1.4.

6.1.1. Initialization Phase

The initialization phase involves starting the Design Server (as described in Appendix H), logging in to the Server via an appropriate Web browser or design tool interface, and selecting an existing design object or naming a new object to work on. Please see Figure 6-2.

6.1.2. Specification Phase

The specification phase deals with the definition and modification of design specifications in the following forms:

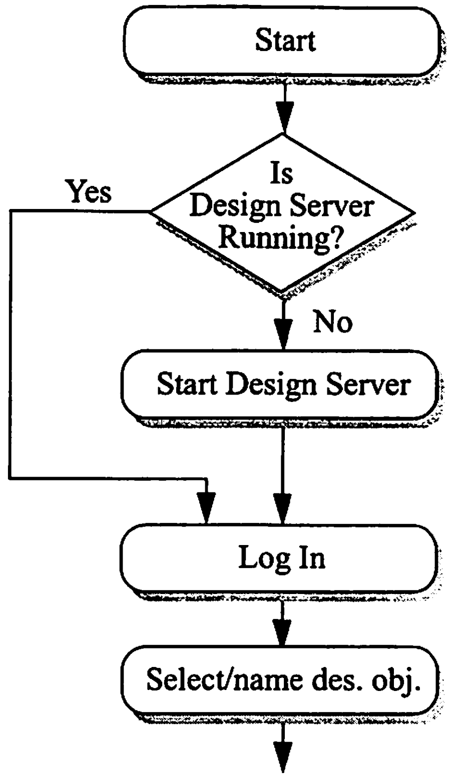


Figure 6-2. Initialization Phase Flow Diagram.

Keywords for Search-based Context Selection - In traditional CAD environments there is no mechanism for exploiting descriptive keywords in the design process. In the information-centric environment, these keywords can be used effectively to locate relevant domains, and subsequently to propose one or more contexts. Descriptive keywords are used primarily in the early design phases where specifications are incomplete or gradually evolving.

Parameters and constraints - Parameters constrain design variables to a single value, while constraints provide bounds for parameters or for acceptable design costs.

Manual Context Manipulation - Due to the importance of selecting a context that accurately represents the dimensions that are relevant to a design, contexts can be controlled directly by designers. Manual control implies that a designer can control the contents of a context on a domain-by-domain basis. The basic mechanism for controlling which domains are in a context requires a designer to manually identify each domain which is to be added to or deleted from a context. A design object can provide a list showing the current domains in its context, plus a list of available domains that can be added. The designer can then add new domains to the context, or delete individual domains from the context. When using the Web browser user interface, adding or deleting domains is as simple as clicking on the appropriate domain in the interface.

As described in Section 5.2.1., when a domain is added to a context, all of its associated domains are added as well. If a designer chooses to delete a domain, it is possible to end up with a non-functional context (e.g., where a domain, containing information used by other domains, is missing). It is left up to the designer to only delete domains for one of the two following purposes. (Note: A design object can avoid ending up with non-functional context by performing a check after context changes but before any other requests are handled. During such a check it can be ensured that all required associated domains are present in a context).

1. To remove a specific domain for the purpose of replacing it with a more generic domain from the same family. This is the way to defeat context creation rule 2 (Section 5.2.1.).

2. To remove all of the domains which use information from the domain to be removed.

The manual context manipulation is useful if a designer has specific domains in mind for a project, or if a context already exists and only needs small adjustments.

Design Files - The first three categories of specifications are either features or side effects of the information-centric design paradigm. The fourth category, design files, is the traditional way to provide detailed design descriptions. Creating and modifying design files proceeds as in the traditional CAD environments, except for cases where domains capture a way to automatically generate design templates based on the given parameters etc. This can ease the burden of creating detailed design descriptions, but will not eliminate it entirely.

6.1.3. Request Phase

Context selection is only a means to an end, and an artifact of the information-centric design model. The goal is to enable designers to interact with the design environment in terms of information exchange, where designers provide specifications and give requests, and where the environment provides feedback in return. The specification and request phases are therefore closely intertwined and designers typically iterate through several specification-request cycles during normal design flows.

6.1.4. Final Phase

The final phase simply involves deciding whether or not to shut down the Design Server, or leave it running until next design session (Figure 6-3). The normal case is to leave it running.

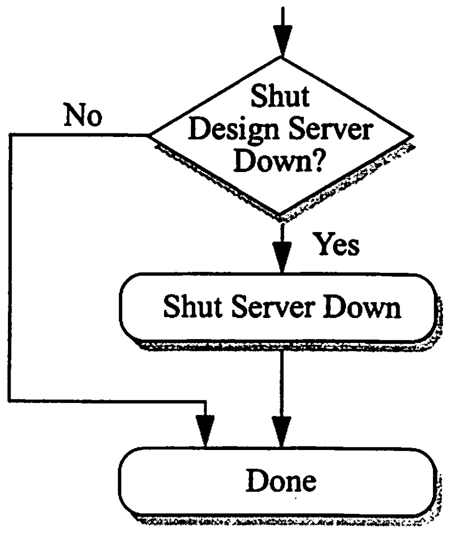


Figure 6-3. Final Phase Flow Diagram.

6.2. Summary

This chapter provided a description of a typical design flow, including the normal iterations that are expected. The following chapter presents four design examples that use the Design Server, to illustrate the benefits of the information-centric design model.

CHAPTER 7

DESIGN EXAMPLES

This chapter presents four design examples to illustrate how an information-centric design environment works. Each of the following sections gives a brief overview of a design scenario, and describes the underlying domains and the design expertise they encapsulate. The domains described in each example have been implemented in the framework of the Design Server.

7.1. CMOS Circuit Design

Design Scenario - Design the transistor schematic for a CMOS NAND gate and find its fall and rise times.

Salient Points - This example shows how the information-centric paradigm simplifies the design and analysis process by reusing encapsulated expertise about how to analyze a digital circuit. It also shows how a designer interacts with the Design Server using simple intuitive commands, and yet benefits from complex analysis tasks.

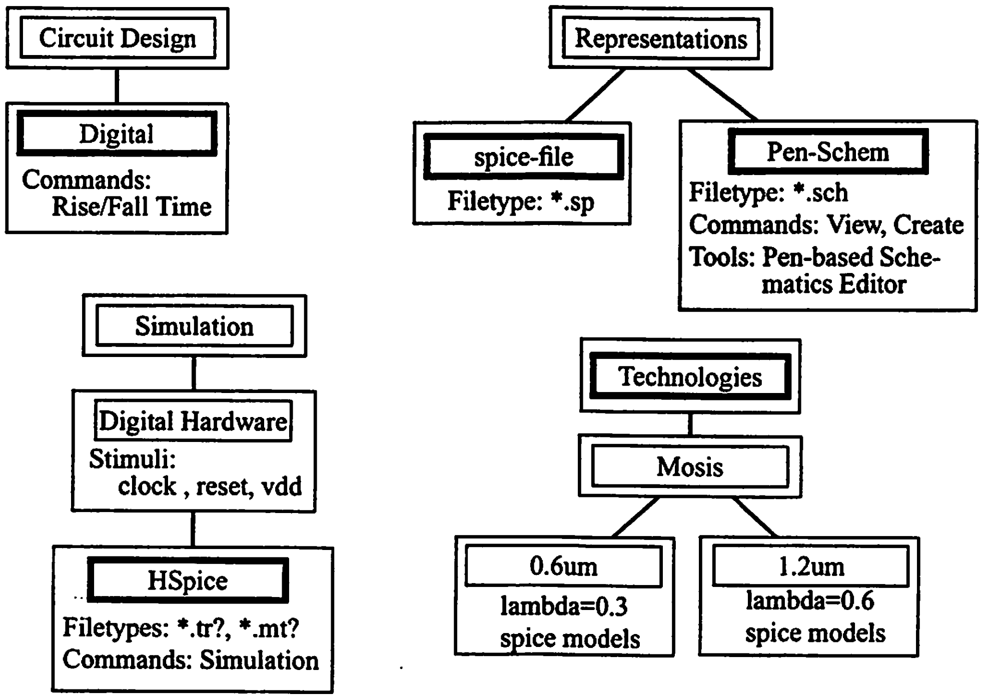


Figure 7-1. Domains for CMOS Circuit Design.

Domains - The domain hierarchies that are used in this example are presented in Figure 7-1. The essential features from each domain are shown, and the domains that are initially selected by the search-based approach are marked by a thick outline.

Design Flow - This scenario follows the steps shown below (*italics* indicate Design Server's actions). Note: This example uses a schematic editor that accepts mouse, pen, and voice input [Narayanaswamy96]. This type of editor is useful in pen-based environments like the InfoPad while other editors may be more convenient in a workstation environment.

1. Log in to the Design Server via a Netscape Web Browser.
2. Give specifications: schematic digital circuit design

Search and propose context "Circuit Design:Digital"

3. Request "Create Schematic," and view the Editor help message that is automatically generated for the benefit of novice users. (see Figures 7-2 and 7-3).

Open mouse/pen/voice-input schematics editor

4. Create schematic for nand-gate and save it.
5. Choose the 0.6 μm technology domain (by default the Generic technology domain is selected).
6. Request "Rise Time." A simulation menu presents which technology is in effect, which Spice model is used, and the values of the parameters "simtime" and "timescale" (these parameters will be explained below). The menu also identifies the default fall and rise times of input signals.

Extract the circuit

Prepare a SPICE deck for simulation

Invoke HSPICE

Extract the requested info from the HSPICE output file (10-90% points)

Present it to the user.

RESULT: 0.87 ns

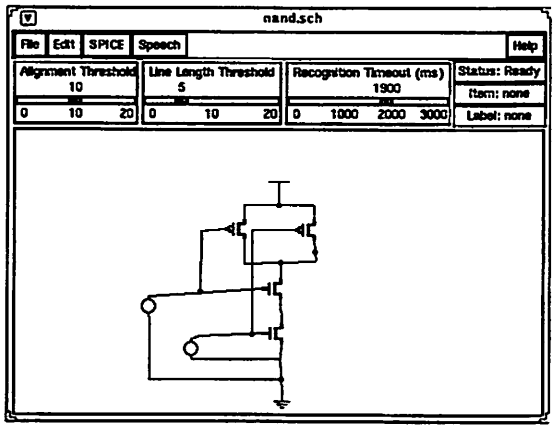
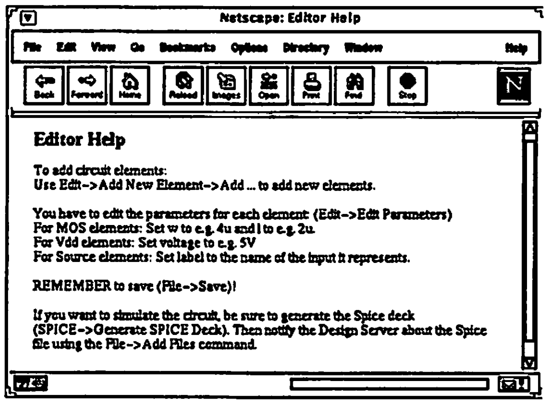


Figure 7-2. Schematic Editor used in example 7.1.



Editor Help

To add circuit elements:
Use Edit->Add New Element->Add ... to add new elements.

You have to edit the parameters for each element: (Edit->Edit Parameters)
For MOS elements: Set w to e.g. 4u and l to e.g. 2u.
For Vdd elements: Set voltage to e.g. 5V
For Source elements: Set label to the name of the input it represents.

REMEMBER to save (File->Save)!

If you want to simulate the circuit, be sure to generate the Spice deck (SPICE->Generate SPICE Deck). Then notify the Design Server about the Spice file using the File->Add Files command.

Figure 7-3. Editor Help Message.

7. Request "Fall Time." Again, a simulation menu is presented, this time showing the values that were entered before. Note: the extraction is not performed again since the circuit has not been modified.

Prepare a SPICE deck for simulation

Invoke HSPICE

Extract the requested info from the HSPICE output file (10-90% points)

Present it to the user.

RESULT: 0.92 ns

Comments The process of analyzing a circuit netlist is tedious and repetitive. It is a necessary step to obtain the desired quantities, but it is not a part of the creative design process. Therefore, it is an immense benefit for a designer to be able to draw upon the encapsulated expertise shown in this example. The points below describe the tasks that are performed automatically, and help illustrate how much effort a designer saves by using the information-centric design environment.

- Find the input signals and the supply voltage. Assumptions: The input signal names can be derived from the names of the voltage sources. For example, if the inputs to the NAND circuit are "A" and "B," there will be voltage sources called VA and VB in the netlist. Likewise, the supply voltage will be named Vdd.
- Let the designer choose appropriate simulation stimuli for each input. Present a list of predefined stimuli, such as "clock," "reset," and the most generic "filereals" which reads values from a given file. The predefined stimuli are encapsulated in the simulation

domains as stimulus templates (see Section 5.1.3.). The HSpice tool encapsulation contains the mappings from the generic stimulus templates to HSpice-specific signals (piecewise linear signals - PWL).

- Generate the proper statements to represent the inputs. The value of the supply voltage (Vdd) is taken from the parameter "voltage" from the technology domain. The piece-wise linear (PWL) specifications are generated from the chosen stimuli for each input. Each transition takes a certain amount of time, given by the parameters "rise_time" and "fall_time" in the HSPICE simulation domain.

```
Vdd vdd 0 5.0
vA A 0 PWL 0 0 1e-8 0 1.1e-8 5 2e-8 5 2.1e-8 0 3e-8 0 3.1e-8 5 4e-8 5 4.1e-8 0
vB B 0 PWL 0 0 1e-8 0 2e-8 0 2.1e-8 5 3e-8 5 3.1e-8 5 4e-8 5 4.1e-8 0
```

- Let the designer identify the names of the output nets, since they cannot be reliably determined from the SPICE netlist. Also, as an option, let the designer specify a loading capacitance to be used to load each output. If the designer specifies a capacitor value, add one capacitor for each output node. For the NAND circuit, there is one output called "S."

```
CS S 0 20f
```

- For each MOS device in the SPICE netlist, specify an appropriate transistor model in accordance with the models that are available in the SPICE model library associated with the chosen technology domain (in this case CMOSN and CMOSP).

```
m1 wire9 B 0 0 CMOSN l=0.6u w=0.6u
m3 S A vdd vdd CMOSP l=0.6u w=1.2u
m2 S A wire9 0 CMOSN l=0.6u w=0.6u
m4 S B vdd vdd CMOSP l=0.6u w=1.2u
```

- Find the SPICE model library file, and refer to it using a ".include" statement:

```
.include `/users/bentz/.designAgent/spice_models/generic/nominal.spm'
```

- Add a statement to measure the rise time:

```
.meas risetime trig par('v(S)-0.1*v(vdd)') val=0 rise=1
+              targ par('v(S)-0.9*v(vdd)') val=0 rise=1
```

- Add the necessary HSPICE statements to the SPICE file. The two values for the ".tran" statement are calculated based on the parameters "simtime" and "timescale" (from the simulation domain hierarchy). "simtime" species the length of the simulation, measured in simulated time, and it is scaled by the value of "timescale." For example, to simulate the first 50 ns of the circuit's transient behavior, set simtime=50 and timescale=1e-9. The second argument for the ".tran" statement is computed by "simtime*timescale" and the first value is computed by "simtime*timescale/100."

```
.options nomod post
.tran 5e-10 5e-08
.end
```

Extensions - This example can be extended to perform other useful analysis tasks such as finding the energy consumption of a digital circuit or perform circuit characterizations for use in cell libraries, etc. In each case, one or more methods have to be created to describe the design flow required, and if additional tools are needed, they have to be encapsulated. To create a method to capture energy consumption analysis, the effort would be relatively small, since the rise and fall time simulations are quite similar to what would be needed. For other cases that may be quite different, the development effort can be moderate to large.

7.2. Finite State Machine Design

Design Scenario - Design a finite state machine (FSM) to control a trafficlight. Estimate the area of two different implementations, one based on a programmable logic array (PLA), the other based on standard cells.

Salient Points - This example shows how the information-centric paradigm facilitates access to remote tools (the FSM editor) and remote files (the FSM design files). It also shows how the command "area" is interpreted differently depending on which domains are in the context.

Domains - The domain hierarchies that are used in this example are presented in Figure 7-4. The essential features from each domain are shown, and the domains that are initially selected by the search-based approach are marked by a thick outline.

Design Flow - This scenario follows the steps shown below (*italics* indicate Design Server's actions). Note: This example uses an FSM editor available on the Web (at http://yoyodyne.EECS.Berkeley.EDU/fsm/fsm.html). It was developed by Wing Yee (Serena) Leung in U.C. Berkeley's WELD group.

1. Log in to the Design Server via a Netscape Web Browser.
2. Give specifications: FSM

   Search and propose context "FSM"

3. Request "Create State Diagram."

   Open Web browser, link to FSM Editor

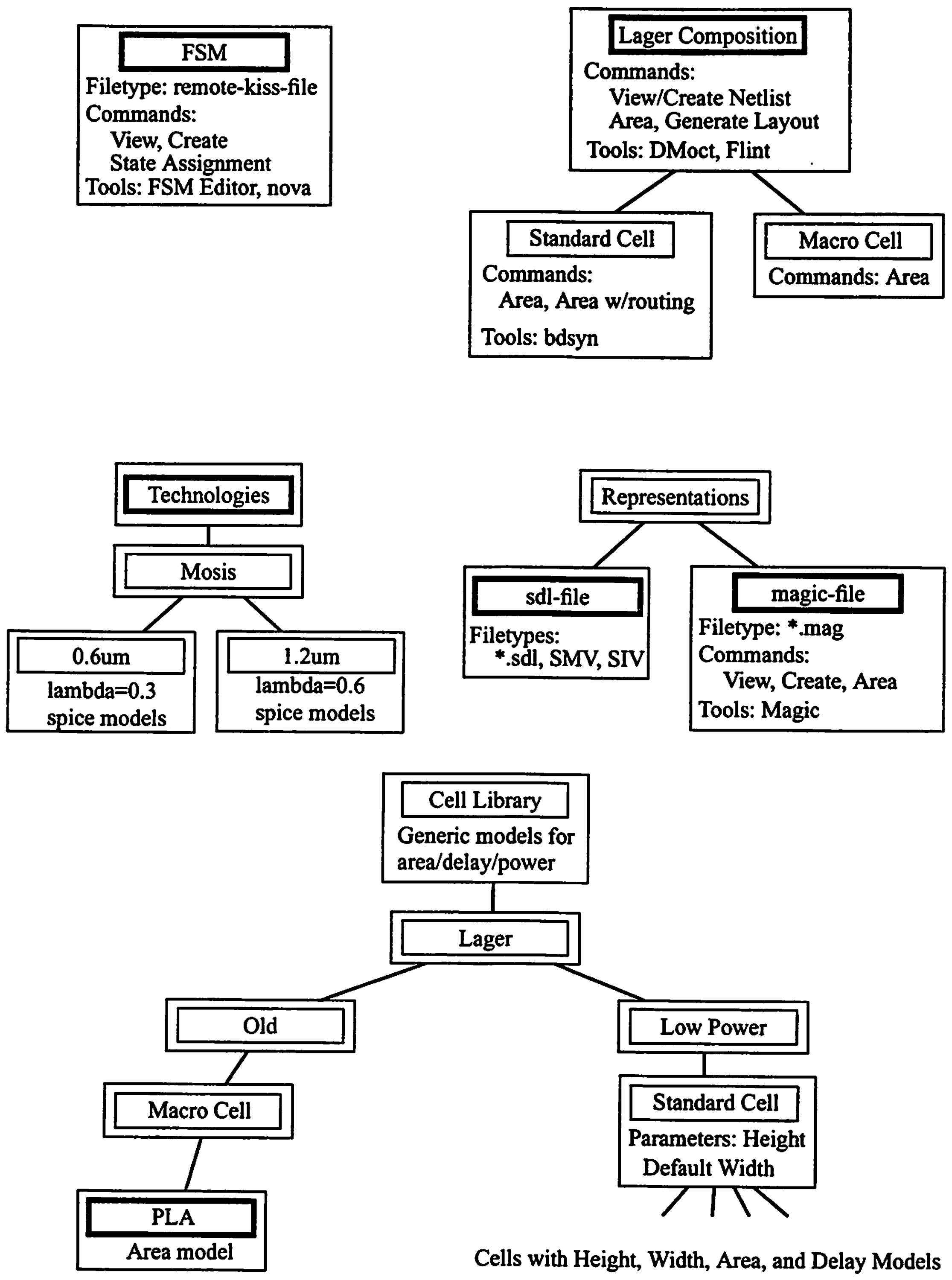
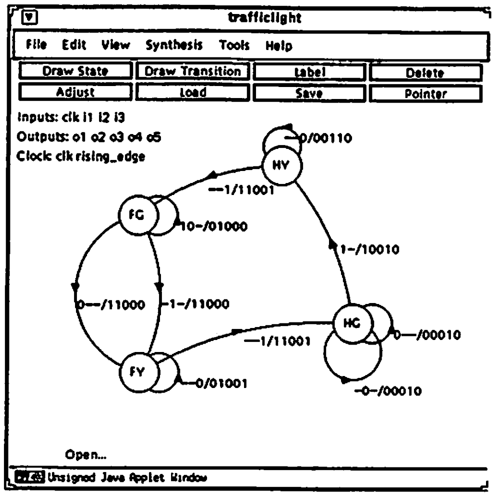


Figure 7-4. Domains for FSM Design.



Inputs: clk i1 i2 i3
Outputs: o1 o2 o3 o4 o5
Clock: clk rising_edge

Figure 7-5. FSM Editor.

4. Draw state diagram and save it as trafficlight.kiss (see Figure 7-5).

5. Request "Area."

Estimate Area based on models in the Macro cell and PLA domains.

RESULT: 88,917 lambda^2

6. Choose the Standard Cell domain from the Lager Low Power Cell library.

7. Request "Area."

Estimate Area based on models in the Standard Cell domains.

RESULT: 142,800 lambda^2

8. Optionally, generate the layout for the two approaches. With the appropriate domains selected, request "Generate Layout." After the layouts have been generated, more accurate area estimates can be obtained: for the PLA style: 91,356 lambda^2, for standard cell style: 115,104 lambda^2. The PLA style area is quite predictable, while the routing overhead of the standard cell implementation is harder to predict.

Comments The information-centric design environment transparently manages local and remote resources. In this example, remote and local tools and files are accessed without any effort on the designer's part. Due to the simple implementation of the remote database (part of the U.C. Berkeley WELD project), there is no way to check file time stamps. Therefore, the Design Server cannot determine when the remote FSM design file has changed. To get around this problem, the following solution was devised. Whenever the remote file is needed as the input to a tool, it is downloaded to the local file system, and compared to the previous version of the file. If no changes have occurred, we keep the previous version, otherwise the new file is used. This solution adds approximately 1.5-2 seconds to the execution time of design tools, but it ensures that any changes to the FSM design can be properly detected.

7.3. Small Processor Example (LagerIV)

Design Scenario - Design the small processor shown in Figure 7-6. Estimate the area after crafting the netlist, and verify the area after generating the layout.

Salient Points - This example shows how the Design Server provides significant aid throughout the design process. It also shows that commands (e.g., "area") remain the same throughout the process, but the interpretation of the commands changes.

Domains - The domain hierarchies that are used in this example are presented in Figure 7-7. The essential features from each domain are shown, and the domains that are initially selected by the search-based approach are marked by a thick outline.

Design Flow - This scenario follows these steps: (*italics* indicate Design Server's actions)

1. Log in to the Design Server via a Netscape Web Browser.

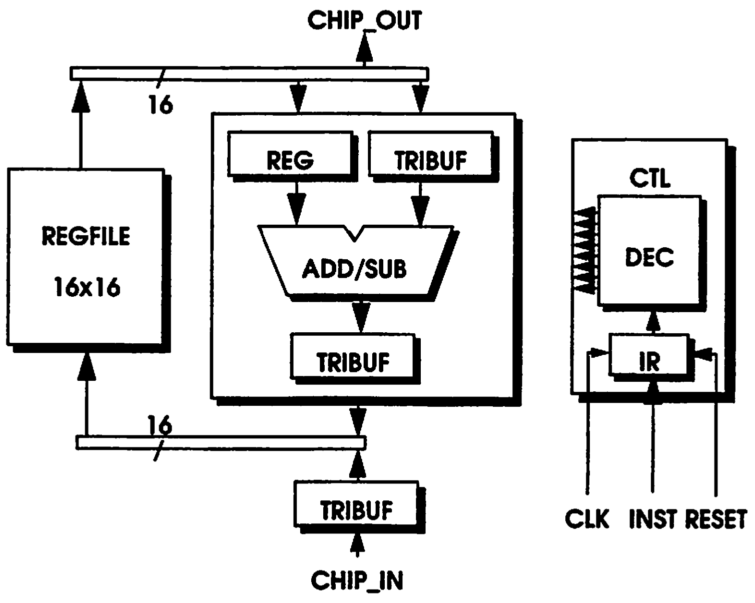


Figure 7-6. Processor Diagram.

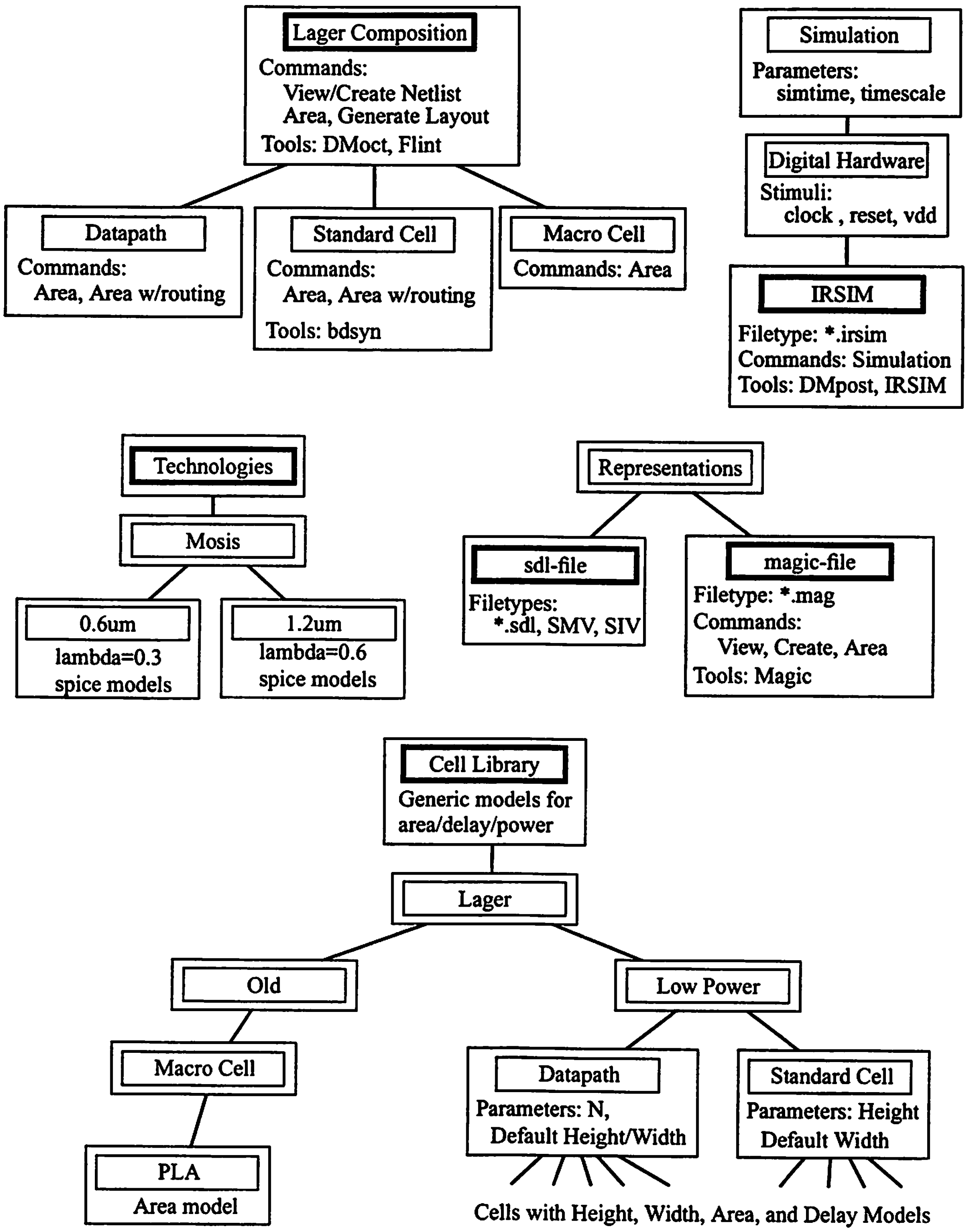


Figure 7-7. Domains for Small Processor Design.

2. Give specifications: ASIC chip design

Search, and Propose the Lager Compositional Domain

3. Optional step: Create block diagram using the HLBE high-level block editor [Olateju96]. Connect to the Design Server and send the design. The use of HLBE enables a designer to get quick estimates of chip area before a detailed netlist is crafted.

4. Request "Area."

Compute Area using models from datapath,

standard cell, and macro cell domains

RESULT: 1,539,859 lambda^2 (no routing or control)

5. Write netlist, save in file "chip.sdl."

6. Add the file "chip.sdl" to the design object.

7. Request "Area."

Parse SDL, create Lager SMV files

Extract essential features from SMV files and build hierarchy of components

Estimate area of each component. Add routing overhead to each block.

RESULT: 2,291,000 lambda^2 (no global routing overhead)

8. Request "Generate VHDL."

Create Lager SIV files, create VHDL representation.

9. Request "View Simulation."

Prepare a command file for simulation

Invoke Synopsys' "vhdlan" VHDL analyzer, then "vhdlsim" simulator.

Show waveforms.

10. Request "Generate Layout."

Create layout using Lager (floorplanning, placement, routing).

11. Request "View Layout."

Show layout file in layout editor (see Figure 7-8).

12. Request "Area."

Extract area from layout

RESULT: 4,377,695 lambda^2

(after manual floorplanning this number drops to 3,599,379 lambda^2).

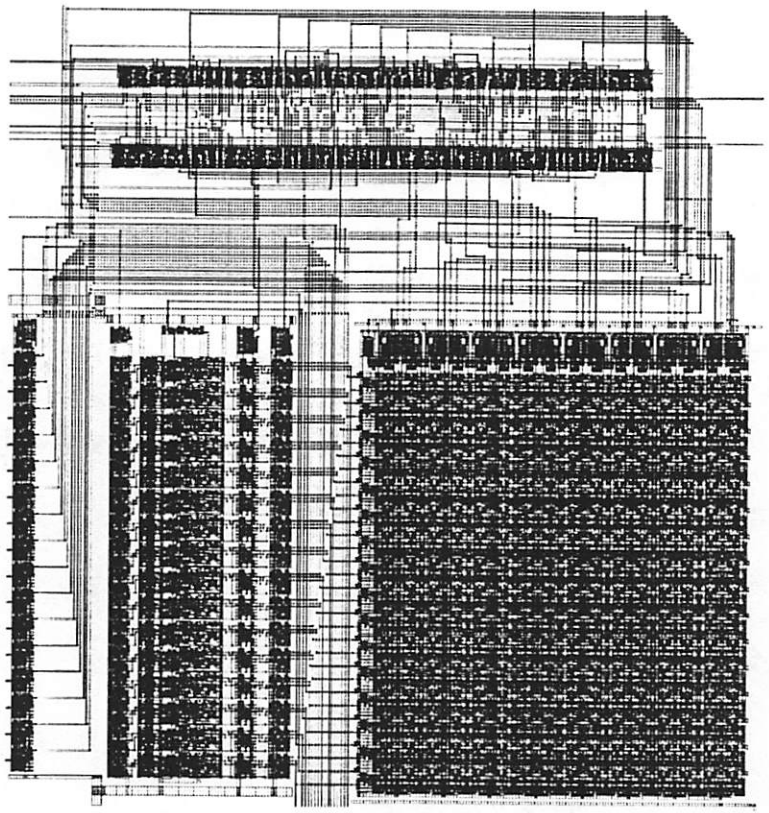
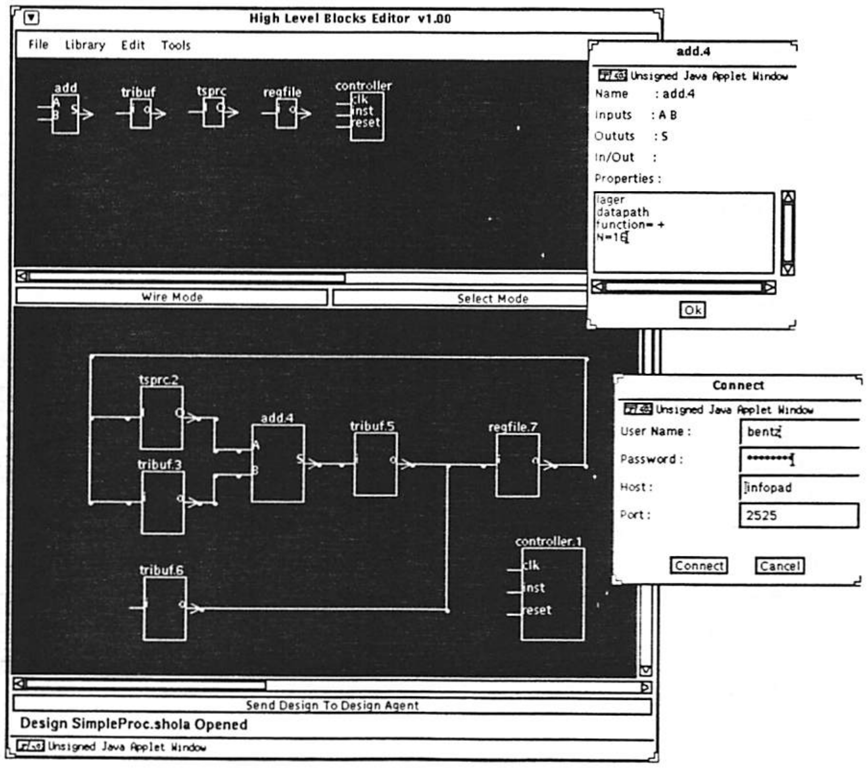


Figure 7-8. Processor Layout.

Comments The use of the HLBE high-level block editor for initial entry of the processor architecture is a quick way to obtain area estimates. The HLBE editor (see Figure 7-9) uses blocks and interconnections to represent an architecture. Each block can be annotated with properties. The properties are used by the Design Server as specifications for each block, and drive the search-based context selection. An HLBE design is not nearly as detailed as a netlist, but it contains enough information to drive the area estimators.

The area estimators in this example span a wide range of estimation techniques. First of all, the techniques can be categorized as either datapath, standard cell, or macro cell (or array cell). Within each of these categories, techniques can model active area or routing, and can be based on default values (e.g., averages for a library), characterizations for spe-



Design SimpleProc.shola Opened

Name : add.4
Inputs : A B
Ututs : S
In/Out :
Properties :

```
lager
datapath
function= +
N=16
```

User Name : bentz
Password : ********
Host : infopad
Port : 2525

Figure 7-9. HLBE Block Editor.

cific cells, values extracted from layouts, or empirical results. Figure 7-10 presents the active area models and shows the underlying structure of each category. Figure 7-11 shows

Category: Datapath

Parameters:

N (number of bits): given in specifications

H (height of bitslice): see below

W (width of bitslice): see below

Model: Active Area = H * W * N

Defaults-based:

H and W: average values for library, given in cell library domain

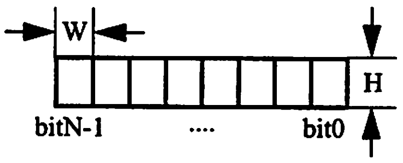
Characterization-based:

H and W: specific values for a cell, given in the domain for each cell

Layout-based Model:

H: extracted from layout, if available

N*W: extracted from layout, if available



Category: Standard Cell

Parameters:

H (height of cell): fixed for a library

W (width of cell): see below

Model: Active Area = H * W

Defaults-based:

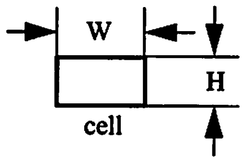
W: average values for library, given in cell library domain

Characterization-based:

W: specific values for a cell, given in the domain for each cell

Layout-based Model:

W: extracted from layout, if available



Category: Macro Cell

Parameters:

N (number of bits): given in specifications

R (number of words): given in specifications

H (height of bitslice): see below

W (width of bitslice): see below

Model: Active Area = H * W * N * R

Defaults-based:

H and W: average values for library, given in cell library domain

Characterization-based:

H and W: specific values for a cell, given in the domain for each cell

Layout-based Model:

H*R: extracted from layout, if available

N*W: extracted from layout, if available

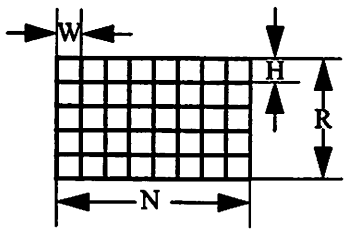


Figure 7-10. Active Area Estimation Models

the overall area models, including routing, used in each category. (NOTE: the models presented here are used in this example. Other models have been presented by other researchers, as outlined in Section 2.4.1.). Currently, there is no routing model for the global level routing.

The active area models are organized such that the characterization-based models are used if available for the cells under consideration, otherwise the layout-based model is used if a layout is available, and finally the defaults-based models are used. This is encapsulated by a method which checks for availability of characterizations and layouts before falling back on defaults. For the area models that include routing, area is extracted from a layout if available, otherwise the models shown in Figure 7-11 are used, after finding the

Category: Datapath

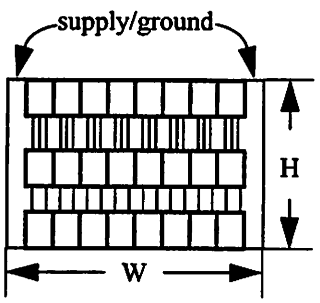
Model: Area = H * W

W (total width) = Max_width_of_subcells + 2 * supply_routing_width

(supply_routing_width = 24 λ)

H (total height) = routing_factor * (Σ height_of_each_subcell)

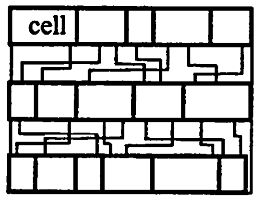
(routing_factor = 1.2 (empirical))



Category: Standard Cell

Model: Area = routing_factor * (Σ Active_area_of_each_cell)

(routing_factor = 2.0 (empirical))



Category: Macro Cell

Model: Area = Active_area

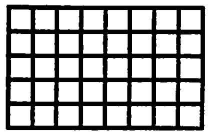


Figure 7-11. Area Estimation Models with Routing.

active area for each of the subcells.

When the request "View Simulation" is given, the Design Server returns a menu in which the user has to specify the following values:

- Time Scaling Factor: timescale = 1e-9
- Simulation Time: simtime = 300
- For Vdd: set monitor = NO
- For clk: set delay = 2 and period = 40
- For reset: delay=1 duration=40
- For GND: set monitor = NO
- For chip_in: set file = chip_in, sample_delay = 44, and sampleperiod = 40
- For inst: set file = inst, sample_delay = 40, and sampleperiod = 40

The data for the input "chip_in" has to be contained by the given file (chip_in). The data for the input "inst" (i.e., the instructions/program) has to be in the file "inst." The simulation results from the VHDL simulation are shown in Figure 7-12. The simulation executes the program:

```
LOAD rf<0>            ; Load the input (1) into rf<0>
LOAD rf<1>            ; Load the input (2) into rf<1>
XFER rf<0>            ; Transfer rf<0> to alu's reg
ADD rf<1>,rf<3>       ; Add alu's reg to rf<1>, and store in rf<3>
STORE rf<3>           ; Put rf<3> on the chip's output
```

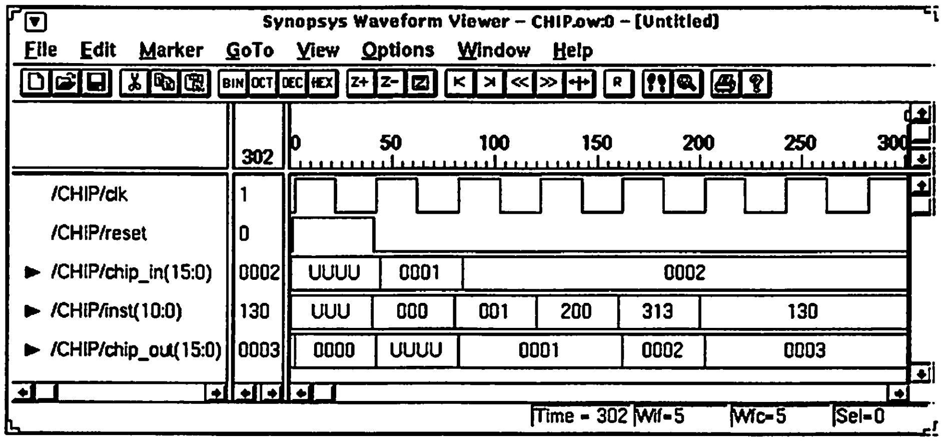

Synopsys Waveform Viewer – CHIP.ow:0 – [Untitled]
File Edit Marker GoTo View Options Window Help
BIN OCT DEC HEX Z+ Z-
302
0 50 100 150 200 250 300
/CHIP/clk 1
/CHIP/reset 0
/CHIP/chip_in(15:0) 0002 UUUU 0001 0002
/CHIP/inst(10:0) 130 UUU 000 001 200 313 130
/CHIP/chip_out(15:0) 0003 0000 UUUU 0001 0002 0003
Time = 302 Wif=5 Wfc=5 Sel=0

Figure 7-12. VHDL Simulation Results.

7.4. Conceptual Level DSP Design

Design Scenario - Design a 1024-point complex Fast Fourier Transform (FFT) algorithm. Compare various alternative FFT forms. Estimate the effect of running the FFT algorithm on different processor architectures.

Salient Points - This example shows how the Design Server facilitates design space exploration at the algorithmic level and at the architectural level.

Domains - The domain hierarchies that are used in this example are presented in Figure 7-13. The essential features from each domain are shown (see also Appendix D).

Design Flow - This scenario follows these steps: (*italics* indicate Design Server's actions)

1. Log in to the Design Server via a Netscape Web Browser.
2. Give specifications: FFT points:=1024

   Search, and Propose the "DSP:DFT:Radix2FFT" Domain

   Set the parameter NrOfPoints to 1024 (based on specification)
3. Request "Complexity."

   Compute complexity using formulae (for number of additions and multiplications)

   RESULT: 51,200 operations
4. Explore two other FFT forms encapsulated by the domains "Radix 4FFT," and "Split Radix." Compare their complexities.

   RESULT: 43,520 operations for Radix4FFT

   RESULT: 34,824 operations for SplitRadix

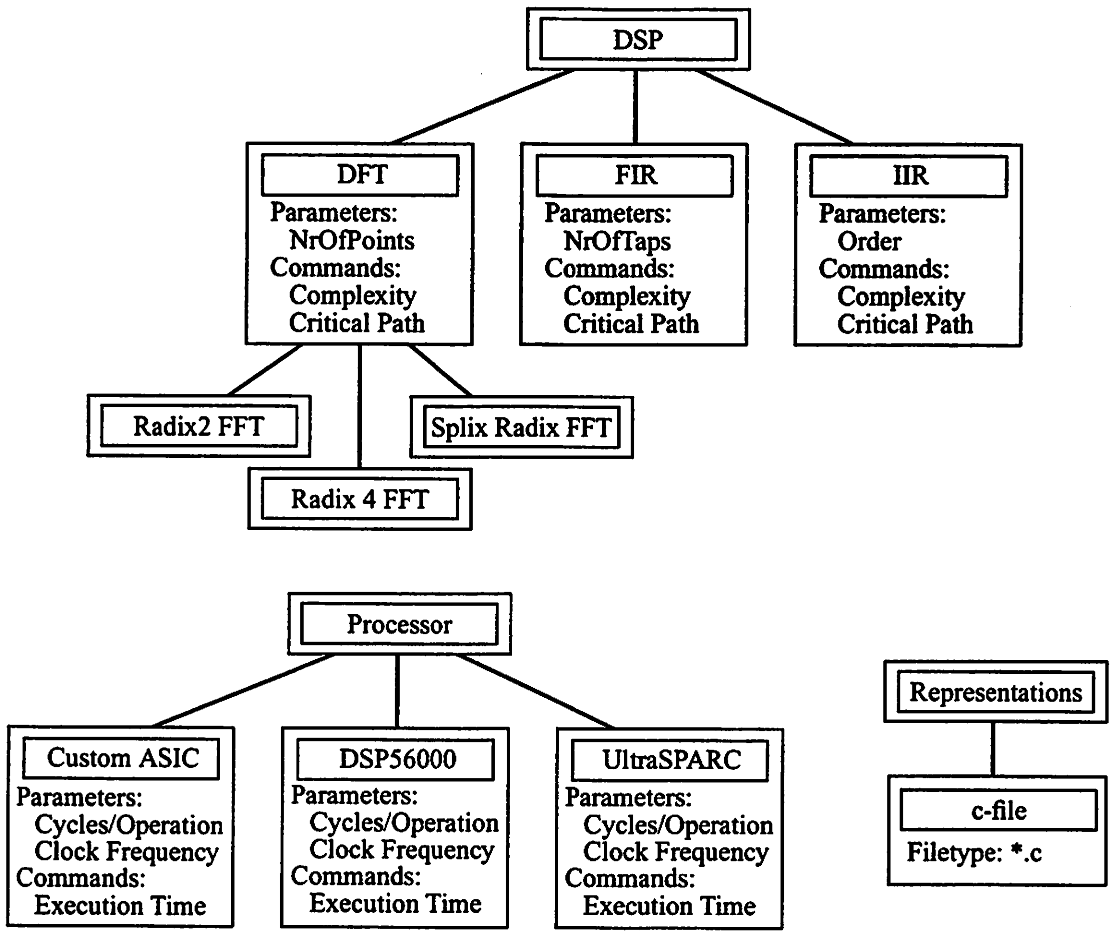


Figure 7-13. Domains for Conceptual DSP Design.

5. Decide to use the SplitRadix FFT algorithm. Choose an implementation platform from the set of available processor domains. The available domains model a digital signal processor (Motorola DSP56K), and a general purpose microprocessor (UltraSparc). Compare their performance.

Compute execution times using formulae that model execution time as a function of algorithmic complexity.

RESULT: 2.28 ms for DSP56K

RESULT: 11.9 ms for UltraSparc

6. To broaden the scope of the exploration, select a completely different implementation

platform, namely a custom ASIC implementation. Again, request the execution time. (Note that in this case execution time is interpreted as the minimum execution time, i.e., the execution time estimate is based on the limitation posed by the algorithmic critical path. This assumes that a large number of parallel computational and storage units are available and does not take into account any area constraints. However, this result shows the minimum possible bound on the execution time, under the assumptions of unlimited available parallel hardware and a 100 ns clock cycle)

RESULT: 8 μs

Comments - It is important to notice the very low level of effort required by the designer to perform this sequence of explorations. The highly heterogeneous set of models for algorithmic complexity and critical path are completely encapsulated and the designer only has to identify which algorithms are of interest. Likewise, the exploration of implementation platform artifacts is equally automated. The complexity of the formulae that correlate the algorithmic complexity or critical path with execution times is completely hidden from the designer, unless he or she is interested and requests to see the assumptions or the details of the models.

The domain definitions for the following domains are given in Appendix D. For algorithmic domains: DSP, DFT, Radix2FFT, Radix4FFT, and SplitRadixFFT. For architectural (processor) domains: GenericArchitecture, DSP56K, and UltraSPARC.

7.5. Summary

The four design examples in this chapter illustrate various of the benefits of the information-centric design model, but it also indirectly illustrates some limitations. The NAND gate example shows the convenience and simplicity of interacting in terms of information. Tedious and repetitive tasks are removed from the designer, such that he/she can focus on the creative work. The finite state machine (FSM) example illustrates the convenience of an environment that can manage local and remote design files and provide seamless access to distributed files and design tools. It also shows how easy it is to perform exploration between alternate design solutions. Initially, quick feedback is provided based on estimation models, but if a designer is willing to spend more time (while the layouts are automatically being generated) accurate feedback can be obtained. The simple processor example shows that the Design Server can provide design aid throughout the design process, including area estimators at various levels of abstraction and automated layout generation. This example also shows the support for integrated simulations, using reusable stimulus generators to generate common signals like clocks, resets, power, and ground. And finally, the FFT example shows how the Design Server supports design exploration of algorithmic alternatives, as well as exploration of how algorithmic properties correlate with architectural properties.

The limitation that is indirectly seen in these examples is that the aid provided by the Design Server is limited by the available domains. Fortunately, designers can create their own domains as necessary, but sometimes this may require significant effort.

CHAPTER 8

FUTURE WORK

The Design Server has been implemented to the level of functionality described in this dissertation, but there are still areas in which it could be extended in interesting ways. Any future extensions has to follow the overall concept of encapsulation, that is, extensions should primarily be encapsulations of design expertise for the purpose of aiding designers. One of the most obvious extensions is to deploy several Design Servers, distributed across a network, and add capabilities to exploit their collective design expertise. Another important area is design guidance, providing qualitative advice upon request or when constraints are violated. These extensions are discussed below, after a retrospective project evaluation that includes other recommendations for future work.

8.1. Project Evaluation

This section highlights both the main strengths of this project as well as its key weaknesses.

8.1.1. Information-centric Paradigm

The information-centric paradigm has proved to be very successful. This paradigm seemed very promising from the beginning of project, and as the project progressed, the information-centric model turned out to be very powerful. From the perspective of designers, the information-centric way of interacting with a CAD environment accelerates numerous tedious tasks, and provides ready access to important design-related information. All these benefits, however, come at the expense of a rich set of domains. The development effort to create good domains requires significant investment in capturing assumptions, creating various levels of estimation models, and providing encapsulations for the tools and methodologies that are typical in an area of expertise. Therefore, it is clear that the usefulness of the information-centric paradigm is in areas where there is a great deal of repetition, i.e., where the investment in domain development yields high returns in terms of time saved and accuracy of results generated as time goes by.

8.1.2. Domain to Domain Interfaces

The adoption of orthogonal domains is another strength of this project. It enables the combination of domains into very diverse contexts with various levels of constraints. It allows designers to provide design specifications ranging from incomplete to highly specific. The downside of dynamically combining domains is that all available domains have to be specifically coded to identify compatibilities and incompatibilities with other domain families. This is not an problem in a local set of domains, but on a grander scale, where domains are created by independent developers and made available through a network, it becomes very difficult to identify all (in)compatibilities. Therefore, I predict that the interface between domains, i.e., the way one domain accesses the information captured in another domain at run-time, will need to be enhanced. Currently, within a context, domains can freely access parameters as long as the exact parameter name is known. This is overly restrictive, since different developers are likely to use different names for similar parameters. Likewise, a method in one domain can freely invoke a method in another domain (that coexists in a context), as long as the method's exact name is known. A better interface between domains would include a query mechanism to allow domains to dynamically determine parameter and method names through a level of indirection. (Example: instead of a method in one domain invoking a method "area" from another domain, it would be better to first be able to query "is there a method in the other domain that can estimate chip area including both active area and routing?" If such a method is not available, a second query could be issued "is there a method in the other domain that can estimate just active area?")

8.1.3. Domain Development Environment

As mentioned above, the effort required to develop good domains is not insignificant. In the Design Server, domains are captured in text files (as shown in Appendix C and D). It would be very useful, and would shorten domain development time significantly, to have a development environment with integral debugging facilities. This would allow incremental development, and could provide support for helping developers capture and codify assumptions. The latter point has proven to be important, especially when different developers create domains that rely on information in other domains. If the assumptions are not clearly identified (and easily accessible) it is very easy to end up with domains that provide invalid design aid or estimation results.

8.1.4. Simulation Support

The automatic simulation capabilities of the Design Server are extremely useful for designers, since it automates the highly tedious and repetitive process of creating a simulation environment for a design, including the proper compilations, stimulus generation, etc. The concept of utilizing generic reusable stimuli is very versatile and allows smooth transitions between simulations at different level of abstraction. However, from a developer's stand point, capturing a simulation methodology has proven to be one of the most complex aspects of domain development. This is one area that would benefit enormously from a good development and debugging environment.

8.1.5. Search-based Context Selection

The search-based context selection mechanism implemented in this project has been quite adequate given the set of currently available domains. One improvement to the current scheme is to allow users to interactively place importance on different types of matches. Currently all matches are considered of equal importance, but at times it may be desirable to only consider parameter matches, or to place a higher importance on matches with domain identifiers.

8.1.6. Domain Versioning

In the current implementation there is no way to specify a version number for a domain. In the distributed Design Server model, it becomes very important that versions of domains can be distinguished, and that old versions of domains are maintained for a certain period of time. By using a versioning scheme, it is possible to identify which generation of a domain to use, and domains on one site can be updated independent of domains on another site. At a later date, the domains on other sites which use the outdated version of a domain can be upgraded to work with the newer version. This is similar to the approach used in Microsoft's Active-X.

8.1.7. Industry Standard Domain Implementation

The current version of the Design Server is implemented in an object oriented version of Tcl. However, to make it easier to proliferate domains, and to solve the domain versioning problem at the same time, it is recommended that domains be implemented in either Corba or Microsoft's Active-X.

8.1.8. Error Calculus

Section 3.4. explained the necessity of providing with an estimate its source (how the estimate was derived) and a measure of its accuracy. In future versions of the Design Server, support for error calculus should be built in, such that when estimates of varying levels of accuracy are combined, a measure of accuracy can be annotated to the final estimate. This requires that results can easily be annotated with accuracy measures. It would make most sense to extend the "sum" (sum over all subcomponents) command to automatically perform the accuracy calculations.

8.2. Distributed Design Servers

System design requires such a broad range of expertise that a designer's lack of breadth or depth critically affects design time and quality. The information-centric design paradigm addresses this by enabling designers to draw upon design expertise in an encapsulated form. It is clear that expertise can not always be found locally, but in many cases has to be retrieved from remote sites. Future Design Servers need to address this by providing the capabilities to enable transparent exploitation of distributed expertise.

The vision is to deploy several Design Servers that are continuously running and serve different categories of clients. The categories that can be foreseen are outlined below (see Figure 8-1).

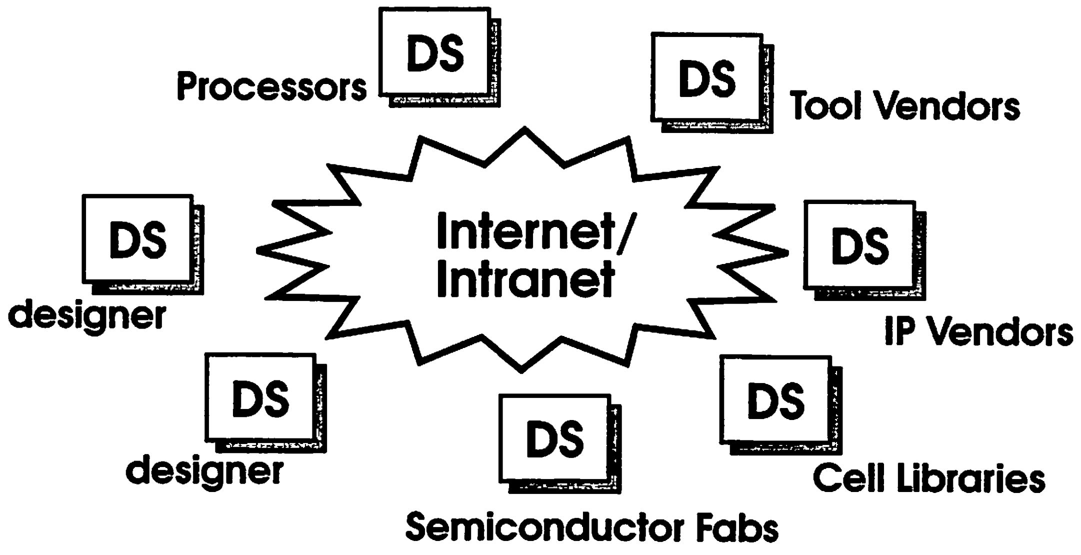


Figure 8-1. Distributed Design Servers.

Designers - will interact with their own Design Server in the same way as with the current Server.

Fabrication Technologies - semiconductor foundries provide characterizations of their technologies, including simulation models (Spice/Hspice), characteristics such as minimum feature size, ring oscillator frequency, design rules, etc.

Cell Libraries - cell library vendors provide an encapsulation of their cells, including characterizations of area, power consumption, delay, etc. They can also provide access to simulation models, schematic diagrams, layouts, etc. Some of the information can be made available for free, while other information may require a contractual agreement with the users.

Intellectual Property Vendors -companies selling IP provide domains that encapsulate information about their products. Intellectual property come in various forms, ranging from algorithms and protocols, to chip cores and ASIC components. In the case of ASIC IP, designs can be made available in various forms, from synthesizable descriptions to final layouts. Independent of where a design lies in this range, the vendor has to provide specifications, characterizations (if applicable), simulation models, documentation, guidance for important design decisions, etc. The Design Server facilitates the transfer of IP by providing an infrastructure for encapsulating a design plus all the additional expertise that must accompany it. A critical issue surrounding IP is property protection. Depending on the nature of the IP, it may be necessary to not disclose certain information. One alternative is to keep sensitive information on the vendor's site where it can be protected, and create interfaces that only give out certain pieces of the information. The vendor can freely give out domains that specify how to access the interfaces to obtain the information a designer needs (see Figure 8-2).

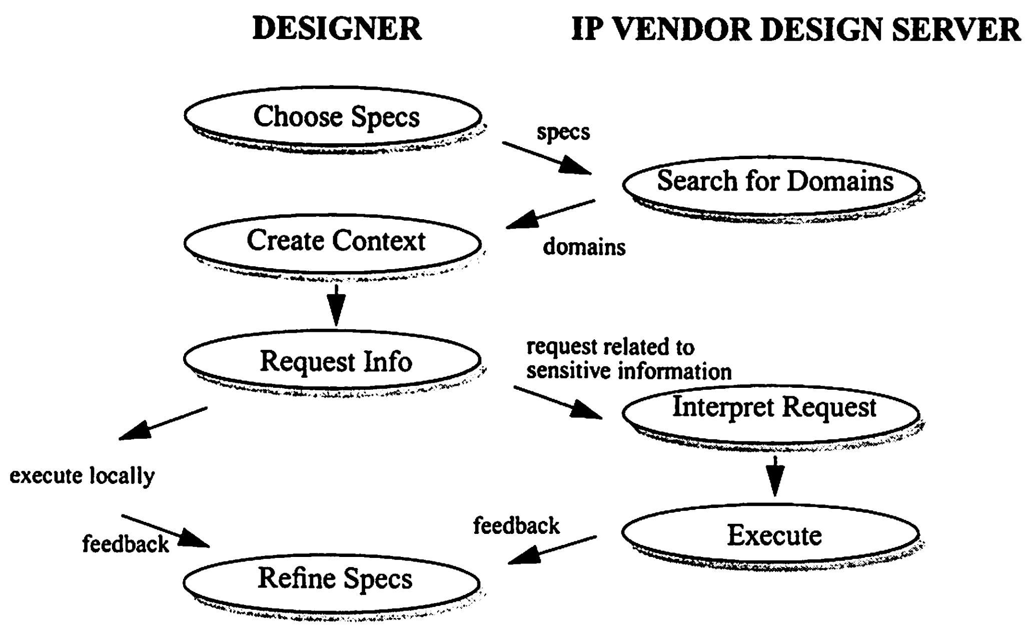


Figure 8-2. Interactions with IP Vendor Servers.

Tool Suites - tool vendors provide encapsulations of their tools, as well as methodologies for using the tools, guidance for manipulating tools, etc. Also, payment models can be provided, either for the service of executing a tool on behalf of a designer, or for licensing the tool to a designer. Models can also be provided, to predict how long it will take to execute a synthesis tool, or to predict the area overhead introduced by a routing tool, etc.

There are some significant challenges that have to be overcome to enable the model of distributed Design Servers. Many of these issues were outlined in Section 3.8. and will be discussed briefly here in relation to this issue.

Finding Design Servers - It is impossible to derive an optimal scheme for how to find Design Servers that have domains of relevance to a given design. However, several pragmatic approaches can easily be devised. To mention a few, a "rolodex" can be maintained, either by each designer, or for a department or company. The rolodex would contain addresses of all known or preferred Servers. Agents or facilitators can also be employed to mediate between design servers and available domains. Also, static name servers (e.g., Yellowpages) and dynamic name servers (e.g., dynamic tool registries) can be employed. It is also useful to maintain a list of Servers that should never be consulted.

Finding Relevant Domains - Once a set of Servers have been located, the next challenge is to determine which Servers have relevant domains for a given design. Design Servers must be able to initiate a search in remote Servers, using the specification driven search mechanism. Once domains have been identified, there has to be a transfer mechanism for bringing a given domain, or a family of domains, to the local Design Server.

Security - In the current version of the Design Server, only users who enter the correct password are allowed access, and when users have access to a Server, they have access to all the domains available to the Server. When servers are allowed to interact with each other, it is necessary to have a richer communication model. Design Servers need to have either an "unprivileged" port through which to talk to other Servers, or an unprivileged mode of access. Furthermore, domains have to be annotated with permissions for reading, to give control over which domains are given out, and which are not. Also, Servers must have a "safe" mode of operation, such that when methodologies from remote domains are executed, they are given only limited privileges, unless the designer designates them as being "trusted."

8.3. Design Guidance

Design guidance is an important aspect of design expertise. Design guidance is qualitative advice about how to manipulate a design for a specific purpose. For example, if a signal processing algorithm does not meet its real-time constraint, design guidance would suggest ways to change the algorithm to make it faster, by retiming, pipelining, etc. As such, guidance is a form of design expertise, and fits nicely in the scope of a domain.

There are a few features that the design server must have to enable the encapsulation of design guidance. First of all, it should be optional whether the Design Server gives guidance when advice on a certain topic is requested, or whenever a constraint has been violated. Secondly, there has to be a mapping between topics and advice, and between constraints and advice. The former is easy, since each piece of advice can be tagged with a list of topics. The latter is slightly more difficult, but the system can be organized such that each constraint can be tagged either with a topic or a guidance script. When a constraint is violated, advice on the given topic can be returned to the designer, or the script can be evaluated and its result returned.

There is not an abundance of specific guidance tools in existence, but when available, such tools can be encapsulated like other design tools. Methodology scripts can also be used in this realm, returning textual advice, or a list of available advice on a given topic. If a specific methodology is known to yield the desired improvements, it can be made available as a part of the advice, such that the designer can invoke it at the “click” of a button.

CHAPTER 9

CONCLUSIONS

This dissertation has presented a new design environment that attempts to bridge the emerging gap between CAD technologies and chip complexities. One of the main contributions of this work is the object-oriented design expertise encapsulation model. This model defines the different types of expertise that need to be captured to aid system designers. The model also provides guidelines for the organization of expertise into relatively independent families.

A second equally important contribution is the model for combining and exploiting domain expertise by creating contexts. Contexts provide a dynamic means for interpreting specific designs, and form the basis for the aid that can be given to designers. By controlling which domains are included in a context, designers can obtain estimates or other design aid under the circumstances and assumptions that are captured in the domains. Contexts also enable an environment to provide aid even at the earliest stages of design, when traditional CAD tools are unable to help. The aid that is provided comes from domains that are selected because they match a set of given design specifications. Thus, when incomplete specifications are given, a context will be assembled from generic domains, and as design specifications are refined, more specific domains will be chosen and used to provide increasingly specific and accurate aid.

The third contribution is the outline of the infrastructure that is needed to implement the information-centric paradigm. Several of the required features are similar to what is found in traditional CAD environments. However, additional capabilities are required to exploit the encapsulated design expertise. The most salient of these is the search engine which locates relevant domains for a particular design. The engine also proposes contexts that provide the best possible match for give specifications. Thus, while the information-centric concept introduces the task of selecting contexts, the search engine minimizes the effort required to do so.

These three elements, the domains, contexts, and the underlying infrastructure, are the core of the solution which we propose to bridge the aforementioned gap. The enormous capacities of chips enable entire systems to be integrated together, spanning a vast range of technologies and specializations. As systems grow larger and more complex, it becomes increasingly important to manage the inherent heterogeneity in methodologies, implementation styles, tools, etc. The compelling need to reduce development time and to increase product performance while building increasingly complex systems calls for a significant increase in the level of abstraction at which designs are conceived and specified. In addition, system level CAD tools must forge strong links between design exploration and implementation paths, to expedite the evaluation of alternative solutions at various stages of the design process.

Previously, there has been very little or no help to be found for meeting these challenges. Therefore, this work is a significant contribution because it presents a solution to these important issues. First of all, it enables designers to deal with a design environment at a higher level of abstraction, namely at the level where information is exchanged (specifications are given, feedback is received). In addition, it can provide aid from the earliest pre-specification design stages, even when specifications are vague and incomplete. Secondly, by using an encapsulation strategy, it integrates the exploration and implementation of designs, while transparently managing the underlying heterogeneity. The encapsulation approach provides a means for exchanging design expertise, resulting in a more efficient design process. For example, design exploration is facilitated by estimation models and techniques that can be executed with minimal input from a designer. Optimization techniques can also be encapsulated to put advanced approaches at the finger tips of system designers. The encapsulation strategy also facilitates the successful transfer of intellectual property, by providing the necessary information about design entities, including documentation, characterizations, how to remotely access the design etc.

The concepts and solutions discussed in this dissertation will only reach their full potential if deployed by a broad spectrum of designers. This is only likely to happen if standards, either *de facto* or official, can be developed for a distributed design infrastructure upon which a future version of the Design Server can be built. Since work is being undertaken in this area (e.g., by the WELD group at Berkeley), advances are likely to be made in the near future. For the more distant future, the information-centric paradigm offers a solution for the compelling needs of systems designers, and as such helps move CAD technology toward closing the gap.

CHAPTER 10

References

[Allen90]	W. Allen, D. Rosenthal, K. Fiduk, "Distributed Methodology Management for Design-in-the-Large," IEEE/ACM International Conference on Computer Aided Design, pages 346-349, 1990.
[Allen91]	W. Allen, D. Rosenthal, K. Fiduk, "The MCC CAD Framework Methodology Management System," Proc. of the 28th ACM/IEEE-CS Design Automation Conference, pages 694-698, 1991.
[Barringer94]	B. Barringer, T. Burd, F. Burghardt, A. Burstein, A. Chandrakasan, R. Doering, S. Narayanaswamy, T. Pering, B. Richards, T. Truman, J. Rabaey, R. Brodersen, "Infopad: A System Design for Portable Multimedia Access", Calgary Wireless 94 Conference, July 1994.
[Beach88]	R.J. Beach, "Documentation Graphics," Proc. Ausgraph 88, Australasian Computer Graphics Assoc., pages 225-230, 1988.
[Ben-Natan95]	R. Ben-Natan, "CORBA: A Guide to Common Object Request Broker Architecture," McGraw-Hill, New York, 1995.
[Bingley92]	P. Bingley, O. ten Bosch, P. van der Wolf, "Incorporating Design Flow Management in a Framework based CAD System", Digest of Technical Papers, IEEE/ACM International Conference on Computer-Aided Design, pages 538-545, 1992.
[Birks90]	G. Birks, P. Davis, "Compound Electronic Document Creation, Storage, Retrieval, and Delivery in Bell-Northern Research and Northern Telecom. Proc. 53rd ASIS Annual Meeting, American Soc. Inf. Sci, pages 16-19, 1990.
[Bosch93]	K.O. ten Bosch, P. van der Wolf, and P. Bingley, "A Flow-Based User Interface for Efficient Execution of the Design Cycle," Digest of Technical Papers, IEEE/ACM International Conference on Computer-Aided Design, pages 356-363, 1993.
[Brockman92]	J.B. Brockman, T.F. Cobourn, M.F. Jacome, S.W. Director, "The Odyssey CAD Framework," IEEE DATC Newsletter on Design Automation, pages 7-11, Spring 1992.
[Brodersen92]	R. W. Brodersen, editor, "Anatomy of a Silicon Compiler," Kluwer Academic Publishers, Boston, Massachusetts, 1992.
[Bushnell89]	M. Bushnell, S.W. Director, "Automated Design Tool Execution in the Ulysses Design Environment," IEEE Trans. on CAD, Vol. 8, No. 3, March 1989.

[CADF91]	CAD Framework Initiative, Inc., "Tool Encapsulation Specification Standard," CFI Doc. DMM-91-G-1, 1991.
[Casotto90]	A. Casotto, R. Newton, A. Sangiovanni-Vincentelli, "Design Management Based on Design Traces," Proc. of the 27th ACM/IEEE-CS Design Automation Conference, 1990.
[Casotto93]	A. Casotto, "Run-Time Requirement Tracing," Digest of Technical Papers, IEEE/ACM International Conference on Computer Aided Design, pages 350-355, 1993.
[Chaudhary95]	K. Chaudhary, M. Pedram, "Computing the Area Versus Delay Trade-off Curves in Technology Mapping," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 14, No. 12, pages 1480-1489, December 1995.
[Chiueh90]	T. Chiueh, R. Katz, "A History Model for Managing the VLSI Design Process," Proc. IEEE International Conference on Computer-Aided Design, pages 358-361, 1990.
[Chokhani94]	S. Chokhani, "Toward a National Public Key Infrastructure," IEEE Communications Magazine, pages 70-74, Sept. 1994.
[Clarke96]	P. Clarke, "WWW primed for EDA," Electronic Engineering Times, n905, June 10, 1996.
[Cmelik93]	B. Cmelik, "Introduction to SpixTools," Sun Microsystems Laboratories, Inc., Mountain View, CA, Revision G, Feb. 1993.
[Crossman92]	R. Crossman, "Standards for Storing Compound Documents," Info DB, Vol. 7, No. 1, pages 17-21, Winter 1992-1993.
[Daniell89]	J. Daniell, S.W. Director, "An Object-Oriented Approach to Distributed CAD Tool Control," Proc. of the 26th ACM/IEEE-CS Design Automation Conference, 1989.
[Dewey89]	A.M. Dewey, S.W. Director, "Yoda: A Framework for the Conceptual Design of VLSI Systems," Proc. of the IEEE Int. Conference on Computer-Aided Design, IEEE, 1989.
[English90]	P.M. English, E.S. Jacobson, R.A. Morris, K.B. Mundy, and others, "An Extensible, Object-oriented System for Active Documents," Proc. International Conference on Electronic Publishing, Document Manipulation and Typography, Cambridge Univ. Press, pages 263-276, 1990.
[Fanderl92]	H. Fanderl, K. Fischer, J. Kamper, "The Open Document Architecture: From Standardization to the Market," IBM Systems Journal, Vol. 31, No. 4, pages 728-754, 1992.
[Gajski88]	D.D. Gajski, editor, "Silicon Compilation," Addison-Wesley Pub. Co., 1988.
[Goldberg83]	A. Goldberg, D. Robson, "Smalltalk-80: The Language and its Implementation," Addison-Wesley, Reading, Mass., 1983.
[Goldfarb90]	C.F. Goldfarb, "The SGML Handbook," Oxford University Press, New York, 1990.
[Granacki83]	J.J. Granacki, A.C. Parker, "The Effect of Register-transfer Design Trade-offs on Chip Area and Performance," Proc. ACM/IEEE 20th Design Automation Conference, pages 419-424, 1983.
[Guerra94]	L. Guerra, M. Potkonjak, J. Rabaey, "System-Level Design Guidance Using Algorithm Properties," IEEE Workshop on VLSI Signal Processing VII, pages 73-82, 1994.
[Guerra96]	L. Guerra, M. Potkonjak, J. Rabaey, "Divide-and-conquer Techniques for Global Throughput Optimization," IEEE Workshop on VLSI Signal Processing IX, pages 137-146, 1996.
[Harrison86]	D.S. Harrison, P. Moore, R. Spickelmier, A.R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment," Proc. IEEE International Conference on Computer-Aided Design, pages 24-27, 1986.
[Hodges88]	D.A. Hodges, H.G. Jackson, "Analysis and Design of Digital Integrated Circuits," McGraw-Hill, New York, 2nd ed., 1988.
[Jacome92]	M.F. Jacome and S.W. Director, "Design Process Management for CAD Frameworks," Proc. of the 29th ACM/IEEE Design Automation Conference, pages 500-505, 1992.

[Jain89] R. Jain, K. Kucukcakar, M.J. Mlinar, A.C. Parker, "Experience with the ADAM Synthesis System," Proc. of the 26th ACM/IEEE Design Automation Conference, pages 56-61, 1989.

[Jain92] R. Jain, A.C. Parker, N. Park, "Predicting System-level Area and Delay for Pipelined and Non-pipelined Designs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 11, No. 8, pages 955-965, August 1992.

[Katz87] R.H. Katz, R. Bhateja, E. Chang, D. Gedye, V. Trijanto, "Design Version Management," IEEE Design and Test, Vol. 4, No. 1, February 1987.

[Kauffman96] S. Kauffman, Jr., J. Perkins, D. Fleet, "Teach Yourself ActiveX Programming in 21 Days," SAMS.NET Publishers, November 1996.

[Kleinfeldt94] S. Kleinfeldt, M. Guiney, J.K. Miller, and M. Barnes, "Design Methodology Management," Proc. of the IEEE, vol. 82, No. 2, pages 231-250, Febr. 1994.

[Kurdahi86] F.J. Kurdahi, A.C. Parker, "PLEST: A Program for Area Estimation of VLSI Integrated Circuits," Proc. 23rd ACM/IEEE Design Automation Conference, pages 467-473, 1986.

[Kurdahi89] F.J. Kurdahi, A.C. Parker, "Techniques for Area Estimation of VLSI Layouts," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, No. 1, pages 81-92, January 1989.

[Landman93] P.E. Landman, J.M. Rabaey, "Power Estimation for High Level Synthesis," Proc. of the European Conference on Design Automation, pages 361-366, 1993.

[Landman94] P.E. Landman, "Low-power Architectural Design Methodologies" Ph.D. Thesis, University of California, Berkeley, December 1994.

[Landman95] P.E. Landman, J.M. Rabaey, "Architectural Power Analysis: The Dual Bit Type Method," IEEE Trans. on Very Large Scale Integration (VLSI) Systems, vol. 3, no. 2, pages 173-187, June 1995.

[Landman96a] P.E. Landman, J.M. Rabaey, "Activity-Sensitive Architectural Power Analysis," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 15, no. 6, pages 571-587, June 1996.

[Landman96b] P.E. Landman, R. Mehra, J.M. Rabaey, "An Integrated CAD Environment for Low-power Design," IEEE Design & Test of Computers, vol. 13, no. 2, pages 72-82, Summer 1996.

[Libes95] D. Libes, "Exploring Expect : a Tcl-based Toolkit for Automating Interactive Programs," O'Reilly & Associates, Sebastopol, CA, 1995.

[Lidsky96] D. Lidsky, J.M. Rabaey, "Early Power Exploration - A World Wide Web Application," Proc. of the 33rd ACM/IEEE Design Automation Conference, pages 27-32, 1996.

[Liu95] P.T. Liu, B. Smith, L. Rowe, "Tcl-DP Name Server," Proc. of the Tcl/Tk Workshop, pages 1-13, 1995.

[Lopez92] J.C. Lopez, M.F. Jacome, and S.W. Director, "Design Assistance for Cad Frameworks," Proc. of the European Conference on Design Automation, pages 494-499, 1992.

[McLennan93] M.J. McLennan, "[incr Tcl]: Object-Oriented Programming in Tcl," Proc. of the Tcl/Tk Workshop, University of California at Berkeley, June 10-11, 1993.

[Mennella90] D. Mennella, A. Muller, "Data Capture of Compound Documents: Solutions to the Problems," Proc. 14th International Online Information Meeting, pages 55-60, 1990.

[MIME92] MIME Internet Standard Document, "RFC 1521", June 1992.

[Nagel75] L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Technical Report UCB/ERL-M520, University of California, Berkeley, May 1975.

[Narayanaswamy96] S. Narayanaswamy, "Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals," Ph.D. Dissertation, University of California at Berkeley, June 1996.

[OCT89]	OCT Manual, U.C. Berkeley EECS Department Technical Report, 1989.
[Ogawa90]	Y. Ogawa, M. Pedram, E.S. Kuh, "Timing-driven Placement for General Cell Layout," IEEE International Symposium on Circuits and Systems, Vol. 2, pages 872-876, 1990.
[Olateju96]	O. Olateju, "Java Based Graphical User Interface For Conceptual Design," Technical Reports by SUPERB Participants, EECS, University of California at Berkeley, 1996.
[Ousterhout94]	J.K. Ousterhout, "Tcl and the Tk Toolkit," Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.
[Park89]	N. Park, R. Jain, A.C. Parker, "Datapath Synthesis of Pipelined Designs: Theoretical Foundations," Progress in Computer-aided VLSI Design: Implementations, Vol. 3, Ablex Publishing, USA, pages 119-156, 1989.
[Parker91]	A.C. Parker, K. Kucukcakar, S. Prakash, Jen-Pin Weng, "Unified System Construction (USC)," High-level VLSI synthesis, Kluwer Academic Publishers, Netherlands, pages 331-354, 1991.
[Pedram91a]	M. Pedram, N. Bhat, "Layout Driven Logic Restructuring/Decomposition," IEEE International Conference on Computer-Aided Design, pages 134-137, 1991.
[Pedram91b]	M. Pedram, N. Bhat, "Layout Driven Technology Mapping," Proc. of the 28th ACM/IEEE Design Automation Conference, pages 99-105, 1991.
[Pedram89]	M. Pedram, B. Preas, "Interconnection Length Estimation for Optimized Standard Cell Layouts," IEEE International Conference on Computer-Aided Design, pages 390-393, 1989.
[Rabaey96]	J.M. Rabaey, "Digital Integrated Circuits: a Design Perspective," Prentice Hall, New Jersey, 1996.
[Rabaey91]	J.M. Rabaey, C. Chu, P. Hoang, M. Potkonjak, "Fast Prototyping of Datapath-Intensive Architectures (the Hyper Synthesis Environment for Real-Time Systems)," IEEE Design & Test of Computers, Vol. 8, No. 2, June, 1991.
[Rabaey85]	J.M. Rabaey, S.P. Pope, R.W. Brodersen, "An Integrated Automated Layout Generation System for DSP Circuits," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-4, No. 3, pages 285-296, July 1985.
[Ruetz86a]	P.A. Ruetz, R. Jain, C. Shung, J.M. Rabaey, and others, "Automatic Layout Generation of Real-time Digital Image Processing Circuits," Proc. of the IEEE 1986 Custom Integrated Circuits Conference, pages 111-115, 1986.
[Ruetz86b]	P.A. Ruetz, S.P. Pope, R.W. Brodersen, "Computer Generation of Digital Filter Banks," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. CAD-5, No. 2, pages 256-265, April 1986.
[Rumbaugh91]	J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-oriented Modeling and Design," Prentice Hall, Inc., New Jersey, 1991.
[Rumsey92]	M. Rumsey and C. Farquhar, "Unifying tool, data and process flow management," Proc. of the European Conference on Design Automation, pages 500-505, 1992.
[Sarma96]	S.E. Sarma, S. Schofield, J.A. Stori, J. MacFarlane, P.K. Wright, "Rapid Product Realization from Detail Design," Computer Aided Design, Vol. 28, No. 5, pages 383-392, May 1996.
[SIA96]	Semiconductor Industry Association, 1996.
[Shung88]	C. Shung, "An Integrated CAD System for Algorithm-specific IC Design," Ph.D. Dissertation, University of California at Berkeley, June 1988.
[Silva93]	M.J. Silva, R.H. Katz, "Active Documentation: A New Interface for VLSI Design," Proc. of the 30th ACM/IEEE Design Automation Conference, pages 654-660, 1993.
[Silva95]	M.J. Silva, R.H. Katz, "The Case for Design Using the World Wide Web," Proc. of the

	32nd ACM/IEEE Design Automation Conference, pages 579-585, 1995.
[Song95]	C. Song, "MIME: Multimedia on the Internet," Unix Review, Vol. 13, No. 4, pages 43-44,46,48-52, April 1995.
[Smith91]	M.D. Smith, "Tracing with pixie," Version 1.1., Center for Integrated Systems, Stanford University, Stanford, CA, April 1991. (Earl Killian of MIPS Computer Systems, Inc. is acknowledged as the author of pixie)
[Stroustrup86]	B. Stroustrup, "The C++ Programming Language," Addison-Wesley, Reading, Mass., 1986.
[Sutton93]	P.R. Sutton, J.B. Brockman, and S.W. Director, "Design Management Using Dynamically Defined Flows," Proc. of the 30th ACM/IEEE-CS Design Automation Conference, pages 648-653, 1993.
[Vaishnav95]	H. Vaishnav, M. Pedram, "Logic Extraction Based on Normalized Netlengths," Proc. International Conference on Computer Design: VLSI in Computers and Processors, pages 658-663, 1995.
[VSI96]	Virtual Socket Interface Alliance, "Virtual Socket Interface Standard," (see http://www.vsi.org/), September, 1996.

APPENDIX A

Extensions to the Tool Command Language (TCL).

The Design Server is implemented in a version of the Tool Command Language (TCL), created by J.K. Ousterhout. This appendix describes the three major packages that I used to extend the capabilities of TCL, in addition to a few commands that I added.

A.1. incr-Tcl

incr-Tcl (pronounced "increment TCL") is created by M.J. McLennan (AT&T) and provides object-oriented extensions to Tcl, much as C++ provides object-oriented extensions to C. The emphasis of incr-Tcl is to support structured programming practices in Tcl without changing the flavor of the language. More than anything else, incr-Tcl provides a means of encapsulating related procedures together with their shared data in a local namespace that is hidden from the outside world. It encourages better programming by promoting the object-oriented "library" mind-set. It also allows for code re-use through inheritance.

In the Design Server, the object-oriented capabilities of incr-Tcl are used to structure and organize significant portions of the system, e.g., the various portions of a domain are implemented as separate classes that are all inherited by the "DOMAIN" class.

A.2. Tcl-DP

Tcl-DP is created by B.C. Smith (Cornell) and L.A. Rowe (U.C. Berkeley). Tcl-DP stands for Tcl Distributed Programming. Tcl-DP adds TCP, UDP, and IP connection management, remote procedure call (RPC), and distributed object support to TCL.

In the Design Server, Tcl-DP is used to implement the network communications channels. Commands from Tcl-DP enable setting up listening sockets that wait for external user interfaces to connect, and facilitates the connection management.

A.3. Expect

Expect is created by D. Libes (NIST). It is a program that "talks" to other interactive programs according to a script. Following the script, Expect knows what can be expected from a program and what the correct response should be. An interpreted language provides branching and high-level control structures to direct the dialogue. In addition, the user can take control and interact directly when desired, afterward returning control to the script.

The name "Expect" comes from the idea of send/expect sequences popularized by uucp, kermit and other modem control programs. However, unlike uucp, Expect is generalized so that it can be run as a user-level command with any program and task in mind. Expect can actually talk to several programs at the same time.

In the Design Server, Expect is used to automatically log in to remote computers, mainly for purposes of monitoring the load levels of a remote computational resource, but can eventually also be used for distributing jobs.

A.4. Other Extensions

I added a few extra commands to TCL to provide some necessary capabilities that either were impossible or difficult to provide as TCL scripts.

fullhostname hostname|IP_addr

> fullhostname takes a hostname or an IP (internet protocol) address (in the standard dot-notation form X.X.X.X) and returns the full, official hostname. This routine is used for the host monitor and for the network access routines.

checkPasswd usename password

> checkPasswd takes a user name and a proposed password, encrypts the password, compares it with the user's entry in the /etc/passwd file, and returns 1 if the password is correct, 0 otherwise. This routine is used to verify a user's identity.

putf [dest_fileid] src_fileid

> putf writes the contents of one file (referred to by a Tcl file identifier, src_fileid)

into another file (referred to by another Tcl file identifier, dest_fileid). If dest_fileid is missing, stdout is used. This routine is typically used for file transfers, especially when files are requested from an external user interface.

posix2date

posix2date converts a POSIX time number (i.e., an integer number of seconds since the "beginning of time") to a standard MIME date string in the format of e.g.

Friday, 01-Mar-96 19:18:04 GMT

This routine is used in the HTML translator.

APPENDIX B

Variables and Commands Available In User-Defined Scripts.

This appendix gives a brief description of the variables and commands that are available for use in the user-defined method scripts.

B.1. VARIABLES

this

The name of the design context in which a method executes.

file

The file object which a method was invoked with.

HistoryIndex

The history index of a tool invocation. This is a number which is chosen by the Design Server. After a tool invocation, the variable HistoryIndex will be set. It gets set even if a tool didn't need to run. The most common use of this variable is to access the stdout (and stderr) of a tool run in the files .stdout$HistoryIndex and .stderr$HistoryIndex.

OUTPUT_FILES

A list of output files that automatically gets set when tools or methods are executed. Users don't usually need access to this variable, except for the types of accesses that can be accomplished with the routine "checkToolOutFile."

RESULT

Users should set this variable to contain a result, if a method is to pass a result to its callers. Often this will be in the form of a Tilt "data" construct (e.g., set RESULT [data area $value $units float]).

DETAIL

This is a list which may be set to allow later retrieval of detailed information about the execution of one or more methods. It is a cumulative list which each successive method can add to, so it is important to use the Tcl command "lappend" when assigning to the DETAIL list. The list can contain Tilt text lists, or Tilt command lists, e.g., { {text "" text to be shown} {command cmd_name cmd args...} ... } The DETAIL variable is initialized when the user issues a command and it can be added to in each subsequent method that gets called.

UNITS

This is a string which the user can optionally set. If it is set, then when other methods are called, they will attempt to return data with values in the given UNITS. This makes it easier to successively call methods without having to worry about doing the unit conversions.

B.2. ROUTINES

Pretty much all the normal Tcl commands are available for use in methods. Two exceptions: Never use the "return" or "exit" commands. If you want to exit the Design Server, use the command "shutdown" which ensures a clean exit.

tool [-force|-probe] TOOL [arg1] [arg2] ... [&]

Invoke a tool called TOOL with the given arguments. The order of the arguments is significant; it must follow the order in which arguments are declared in the tool definition. Normally, the "tool" command will only result in an actual tool execution if necessary, i.e., if it hasn't been executed since the last change in the status of the tool's inputs and outputs. However, the flag "-force" can be used to force a tool to rerun. This is often needed to make sure certain tools execute every time they are called upon (e.g., xgraph). tool returns the string "SUCCESS," or an error message that begins with "FAILURE:". If the "-probe" flag is used, tool never executes, but returns "NO NEED" or "NEED" depending on whether the tool "needs" to be run. NOTE: The return value from tool runs may be revised soon. If the last argument is the "&" the tool will be executed as a background process.

check string

This routine checks a string which normally comes from a tool run (e.g., check [tool TOOL1]). If the string reads "SUCCESS," everything moves on normally. If the string does not read "SUCCESS," an error message is displayed. NOTE: This routine may become obsolete.

checkToolOutFile index errorMsg

This routine verifies that the variable OUTPUT_FILES exists, and that it has at least $index+1 elements in it. It then returns the name of the index'th file object in the OUTPUT_FILES list. If an error occurs, then the errorMsg is used to identify which tool has caused the problem.

run method [FILEOBJ]

Run a method. If a file object is given, it will be pass to the method in the variable "file." If a file object is not given, the "run" routine attempts to determine what to do: If a method if defined for the "hierarchy" type, AND if the design object is hierarchical, that method will be used. Otherwise, if a method is defined for the "nil" type, it will be used. Otherwise, the Design Server will look for files of the types for which methods have been defined, and let the user resolve any ambiguities.

baseName FILEOBJ|filename

Return the base of the current name of a file object, or of a filename. The base it what remains when the leading path component AND the trailing .extension are removed. For example, "baseName /tmp/hi.txt" returns "hi".

subcells [add designObj1 ....]

Returns a list of the subcells that belongs to the current design context, or add design objects to the subcell list.

getParameter

Returns a list of all "name and value" parameter pairs.

getParameter parameter_name

Returns the current value of the given parameter.

setParameter name value

Sets the parameter 'name' to the value 'value'.

files fileobj1 [fileobj2] ...

Add the given file objects to the current context.

files [-type filetype]

Return a list of all the files that belong to the context. If the -type flag is used, return only files that are of the given file type.

context

context add [-after domainN] [-before domainN] domain1 [domain2] ...

context delete domain1 [domain2] ...

Show, add, or delete domains from the context.

DesignObject name

Create a design object by the given name. The name cannot be the same as any existing command.

File name

Create a file object, and give it the name. Return the name of the file object, which will be FILE%d (where %d is an integer).

sum command units list_of_subcells

Return a Tilt data containing the sum of each of the values returned from a "run $command" call to each of the listed subcells. The values are converted to "units" before being added. Sum automatically creates a DETAIL entry that lists each of the values that were added.

max command units list_of_subcells

Return a Tilt data containing the maximum of each of the values returned from a "run $command" call to each of the listed subcells. The values are converted to "units" before being compared.

min command units list_of_subcells

Return a Tilt data containing the minimum of each of the values returned from a "run $command" call to each of the listed subcells. The values are converted to "units" before being compared.

convertUnits value units new_units

Returns the value converted from units to new_units. For example, convertUnits 11 mW W would return "0.011". If a conversion cannot be done, i.e., a specification for the requested conversion cannot be found, an empty string is returned.

getPath filename

Returns the directory path to the given filename, or an empty string if the filename could not be found along the search path. To get the full path name you can do:

```
set fullpath "[getPath $filename]/$filename"
```

This command is useful for locating helper scripts or application specific files.

APPENDIX C

Domain Template

This appendix contains a template for creating domains. The syntax of each category of declarations is shown in comment lines (beginning with a # character). This template helps users to develop syntactically correct domain definitions.

```
#
# Template --
#
#
# Author: Ole Bentz, 1996
#
Domain Template -inherit DOMAIN {
    ############################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ############################################################################

    ############################################################################
    #
    # Default Domains
    #
    # domains domain1 [domain2] ...
    # domains -req domain1 [domain2] ...
    #
    ############################################################################
```

```
##########################################################################
#
# Search Path
#
# path path1 [path2] ...
#
##########################################################################

##########################################################################
#
# Parameters
#
# parameter name [-group menuGroup] -varname varname -value value \
#  [-type type] [-keep] [-propagate]
#
##########################################################################

##########################################################################
#
# Constraints
#
# constraint name [-group menuGroup] [-keep] [-ignore]
#  and either of these:
#    -parameter name -min value
#    -parameter name -max value
#    -parameter name -min value -max value
#    -parameter name -values list_of_values
#    -method name [-units units] -min value
#    -method name [-units units] -max value
#    -method name [-units units] -min value -max value
#    -method name [-units units] -values list_of_values
#    -include_domains list_of_domains
#
##########################################################################

##########################################################################
#
# Abstract Commands
#
# command name [-group name] [-method method -filetypes list_of_filetypes]
#   where command is one of
#     view
#     analysis
#     optimization
#     action
#
##########################################################################

##########################################################################
#
# Filetypes
#
# filetype name -icon path -name_patterns list_of_patterns \
#  -contents {offset string} -attributes list_of_attributes
#
##########################################################################
```

```
    ###########################################################################
    #
    # Tool Definitions
    #
    # tool name -input input_spec_list [-input input_spec_list] ... \
    #  -output output_spec_list [-output output_spec_list] ... \
    #  -stimulus_port {-var val [-var val] ... script} \
    #  -clusters list_of_clusters -environment env_specs \
    #  -execute exec_script -status status_script
    #
    ###########################################################################

    ###########################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ###########################################################################

    ######################################################################
    #
    #  STIMULUS NUMBER GENERATOR DEFINITIONS
    #
    # numgen name [-parameter_name default_value] \
    #   [-parameter_name default_value] ... script
    #
    ######################################################################

    ###########################################################################
    #
    # Unit Conversions
    #
    # unit_conversion type1 type2 script_from_1_to_2 script_from_2_to_1
    #
    ###########################################################################

    ###########################################################################
    #
    # Translations
    #
    #  Scaling, extrapolation, interpolation, etc.
    #
    # translation label script
    #
    ###########################################################################

}
```

APPENDIX D

Domain Examples

This appendix contains the definitions for the domains used in Section 7.4. The domains, in the order they are listed below, are: DSP, DFT, Radix2FFT, Radix4FFT, SplitRadixFFT, GenericArchitecture, DSP56K, and UltraSPARC.

```
#
# DSP --
#
Domain DSP {
    ###########################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ###########################################################################
    key domaintype dsp "digital signal processing" "signal processing"

    ###########################################################################
    #
    # Default Domains
    #
    # domains domain1 [domain2] ...
    #
    ###########################################################################
    domains {Representations:C}
    domains {Representations:Silage}

    ###########################################################################
    #
    # Parameters
    #
    # parameter name [-group menuGroup] -varname varname -value value \
    #  [-type type] [-keep]
    #
    ###########################################################################
    parameter "Sample Frequency (Hertz)" -varname sampleRate -value 1.0e6 \
          -type float -keep
}
```

```
#
# DFT --
#
# Model for N^2 DFT routine (definition of DFT)
#
# Author: Steven Chan, Nov 1996
#
Domain DFT -inherit DSP {
    ##########################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ##########################################################################
    key domaintype dft "discrete fourier transform"

    ##########################################################################
    #
    # Parameters
    #
    # parameter name [-group menuGroup] -varname varname -value value \
    #  [-type type] [-keep] [-propagate]
    #
    ##########################################################################
    parameter "Number of points" -varname NrOfPoints -value 1024 \
           -type integer -keep

    ##########################################################################
    #
    # Abstract Commands
    #
    # command name [-group name] [-method method -filetypes list_of_filetypes]
    #   where command is one of
    #     view
    #     analysis
    #     optimization
    #     action
    #
    ##########################################################################
    view Properties
    view "Critical Path" -group Properties -method critical_path -filetypes {nil}
    view "Complexity" -group Properties -method complexity -filetypes {nil}
    view "Memory Requirement" -group Properties -method memory -filetypes {nil}

    ##########################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ##########################################################################
    #
    # Three generic routines to be reused in the FFT derivatives.
    #
    meth memory nil {
      set N [run memory_model nil]
      set RESULT "Approximate memory storage requirements: $N words\n"
      append RESULT "Memory order of growth: O(N)"
   }
   meth critical_path nil {
      set operationList [run critical_path_model nil]
      set path 0
      foreach operation $operationList {
          incr path [lindex $operation 1]
          lappend breakdown \
          "Number of [lindex $operation 0] : [lindex $operation 1] ; "
      }
      set RESULT [data "Critical Path" $path Operations integer]
      lappend DETAIL [text "BREAKDOWN" [join $breakdown "\n"]]
   }
   meth complexity nil {
      set operationList [run operation_count_model nil]
      set count 0
      foreach operation $operationList {
          incr count [lindex $operation 1]
          lappend breakdown \
            "Number of [lindex $operation 0] : [lindex $operation 1] ; "
      }
      set RESULT [data Complexity $count Operations integer]
      lappend DETAIL [text "BREAKDOWN" [join $breakdown "\n"]]
   }
   #
   # Three specific routines for this DFT.
   #
   meth memory_model nil {
      set RESULT [expr 2*$NrOfPoints]
   }
   meth critical_path_model nil {
      set N $NrOfPoints
      lappend RESULT [list "real +" [expr 4*$N - 2]]
      lappend RESULT [list "real *" [expr 4*$N]]
   }
   meth operation_count_model nil {
      set N $NrOfPoints
      set 4_N_square [expr int(4*$N*$N)]
       lappend RESULT [list "real +" [expr int($4_N_square-2*$N)]]
       lappend RESULT [list "real *" $4_N_square]
   }
}
```

```
#
# Radix2FFT --
#
# Model for Radix 2 FFT Algorithm
#
# Author:     Steven Chan, November 1996
# Updated by: Ole Bentz, December 1996
#
Domain Radix2FFT -inherit DFT  {
    ############################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ############################################################################
    key domaintype fft "fast fourier transform" radix2fft radix "radix 2"

    ############################################################################
    #
    # Constraints
    #
    # constraint name [-group menuGroup] [-keep] [-ignore]
    #  and either of these:
    #    -parameter name -min value
    #    -parameter name -max value
    #    -parameter name -min value -max value
    #    -parameter name -values list_of_values
    #    -method name [-units units] -min value
    #    -method name [-units units] -max value
    #    -method name [-units units] -min value -max value
    #    -method name [-units units] -values list_of_values
    #    -include_domains list_of_domains
    #
    ############################################################################
    constraint "Number of points" -parameter NrOfPoints -values \
       {2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 \
        32768 65536 131072 262144 5242880 1048576 2097152 4194304}

    ############################################################################
    #
    # Abstract Commands
    #
    # command name [-group name] [-method method -filetypes list_of_filetypes]
    #   where command is one of
    #     view
    #     analysis
    #     optimization
    #     action
    #
    ############################################################################
    view "Visualize Algorithm" -group Documentation -method visualize -filetypes
    {nil}

    ############################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ###########################################################################
    meth visualize nil {
       set script_name "[getPath fft_visualization]/fft_visualization"
        set N [getParameter NrOfPoints]
        set algo_name "Radix_2"
       if {$N > 128} {
          set RESULT "FAILURE:\nCannot display more than 128 point FFT"
       } else {
              set status [exec $script_name $algo_name $N &]
              if {$status != 0} {
                  set RESULT "SUCCESS"
              } else {
                  set RESULT "FAILURE:\nErrors occurred"
              }
       }
    }
    meth critical_path_model nil {
       set N $NrOfPoints
       set log2N [expr log($N)/log(2)]
       set Mc [expr int($log2N)]
       set Ac [expr int($log2N)]
        lappend RESULT [list "real +"  [expr 2*$Ac + 2*$Mc]]
        lappend RESULT [list "real *"  [expr 4*$Mc]]
    }
    meth operation_count_model nil {
       set N $NrOfPoints
       set log2N [expr ceil(log($N)/log(2))]
       #
       # Calculate the number of complex multiplications and additions
       #
       set Mc [expr ceil($N/2.0) * $log2N]
       set Ac [expr $N * $log2N]
       #
       # Calculate the number of real multiplications and additions
       #
       set Mr [expr int(4*$Mc)]
       set Ar [expr int(2*$Ac + 2*$Mc)]
        lappend RESULT [list "real +" $Ar]
        lappend RESULT [list "real *" $Mr]
    }
}
```

```
#
# Radix4FFT --
#
# Model for Radix 4 FFT Algorithm
#
# Author:     Steven Chan, November 1996
# Updated by: Ole Bentz, December 1996
#
Domain Radix4FFT -inherit DFT  {
    ###########################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ###########################################################################
    key domaintype  fft "fast fourier transform" radix4fft radix "radix 4"

    ###########################################################################
    #
    # Constraints
    #
    # constraint name [-group menuGroup] [-keep] [-ignore]
    #  and either of these:
    #    -parameter name -min value
    #    -parameter name -max value
    #    -parameter name -min value -max value
    #    -parameter name -values list_of_values
    #    -method name [-units units] -min value
    #    -method name [-units units] -max value
    #    -method name [-units units] -min value -max value
    #    -method name [-units units] -values list_of_values
    #    -include_domains list_of_domains
    #
    ###########################################################################
    constraint "Number of points" -parameter NrOfPoints -values \
       {4 16 64 256 1024 4096 16384 65536 262144 1048576 4194304}

    ###########################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ###########################################################################
    meth critical_path_model nil {
       set N $NrOfPoints
       set log4N [expr log($N)/log(4)]
       set Mc [expr int($log4N)]
       set Ac [expr 2 * int($log4N)]
         lappend RESULT [list "real +" [expr 2*$Ac + 2*$Mc]]
         lappend RESULT [list "real *" [expr 4*$Mc]]
    }
    meth operation_count_model nil {
       set N $NrOfPoints
       set log2N [expr ceil(log($N)/log(2))]
```

```
        #
        # Calculate the number of complex multiplications and additions
        #
        set Mc [expr ceil(3*$N/8.0) * $log2N]
        set Ac [expr $N * $log2N]
        #
        # Calculate the number of real multiplications and additions
        #
        set Mr [expr int(4*$Mc)]
        set Ar [expr int(2*$Ac + 2*$Mc)]
         lappend RESULT [list "real +" $Ar]
         lappend RESULT [list "real *" $Mr]
    }
}
```

```
#
# SplitRadix --
#
# Model for a Split-Radix FFT Algorithm
#
# Author:     Steven Chan, November 1996
# Updated by: Ole Bentz, December 1996
#
Domain SplitRadix -inherit DFT  {
    ##############################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ##############################################################################
    key domaintype  fft "fast fourier transform" "Split Radix"

    ##############################################################################
    #
    # Constraints
    #
    # constraint name [-group menuGroup] [-keep] [-ignore]
    #  and either of these:
    #     -parameter name -min value
    #     -parameter name -max value
    #     -parameter name -min value -max value
    #     -parameter name -values list_of_values
    #     -method name [-units units] -min value
    #     -method name [-units units] -max value
    #     -method name [-units units] -min value -max value
    #     -method name [-units units] -values list_of_values
    #     -include_domains list_of_domains
    #
    ##############################################################################
    constraint "Number of points" -parameter NrOfPoints -values \
       {2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 \
        32768 65536 131072 262144 5242880 1048576 2097152 4194304}

    ##############################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ##############################################################################
    meth critical_path_model nil {
       set N $NrOfPoints
       set log2N [expr log($N)/log(2)]
       set Mc [expr int($log2N)]
       set Ac [expr int($log2N)]
         lappend RESULT [list "real +" [expr 2*$Ac + 2*$Mc]]
         lappend RESULT [list "real *" [expr 4*$Mc]]
    }
    meth operation_count_model nil {
       set N $NrOfPoints
```

```
        set log2N [expr ceil(log($N)/log(2))]
        #
        # Don't bother with the number of complex mults and adds.
        # Calculate the number of real multiplications and additions
        #
        set Mr [expr int($N*($log2N-3) + 4)]
        set Ar [expr int(3*$N*($log2N-1) + 4)]
         lappend RESULT [list "real +" $Ar]
         lappend RESULT [list "real *" $Mr]
    }
}
```

```
#
# GenericArchitecture Processor Model
#
# Author: Steven Chan, Nov 1996
#
Domain GenericArchitecture -inherit Generic {
    ############################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ############################################################################
    key domaintype architecture processor cpu

    ############################################################################
    #
    # Parameters
    #
    # parameter name [-group menuGroup] -varname varname -value value \
    #  [-type type] [-keep] [-propagate]
    #
    ############################################################################
    parameter "Has Floating Point Unit?" -varname hasFlPtUnit -value 1 -type int
    parameter "Clock Rate (Hertz)" -varname cycle_rate -value 100E6 -type float
    #
    # Parameters describing the dynamic nature of programs to be executed
    # on this processor:
    #
    parameter "Dynamic Instruction Count" -group "Inst Dyn" \
      -varname dyn_inst_cnt -value 0 -type int
    parameter "Branch Frequency" -group "Inst Dyn" \
      -varname freq_branch -value 0.0 -type float
    parameter "FP Add Frequency" -group "Inst Dyn" \
      -varname freq_fp_add -value 0.0 -type float
    parameter "FP Mul Frequency" -group "Inst Dyn" \
      -varname freq_fp_mul -value 0.0 -type float
    parameter "FP Div Frequency" -group "Inst Dyn" \
      -varname freq_fp_div -value 0.0 -type float
    parameter "Int ALU, Logic, Shift Frequency" -group "Inst Dyn" \
      -varname freq_int -value 0.0 -type float
    parameter "Int Mul Freq" -group "Inst Dyn" \
      -varname freq_int_mul -value 0.0 -type float
    parameter "Int Div Frequency" -group "Inst Dyn" \
      -varname freq_int_div -value 0.0 -type float
    parameter "Load/Store Frequency" -group "Inst Dyn" \
      -varname freq_mem -value 0.0 -type float
    #
    # Characteristics of the processor:
    #
    parameter "Cycles Per Branch" -group "Cycle Times" -varname cycles_branch \
      -value 2.0 -type float
    parameter "Cycles Per FP Add" -group "Cycle Times" -varname cycles_fp_add \
      -value 2.0 -type float
    parameter "Cycles Per FP Mul" -group "Cycle Times" -varname cycles_fp_mul \
      -value 3.0 -type float
```

```
parameter "Cycles Per FP Div" -group "Cycle Times" -varname cycles_fp_div \
  -value 5.0 -type float
parameter "Cycles Per Int ALU, Logic, Shift" -group "Cycle Times" \
  -varname cycles_int -value 1.0 -type float
parameter "Cycles Per Int Mul" -group "Cycle Times" \
  -varname cycles_int_mul -value 3.0 -type float
parameter "Cycles Per Int Div" -group "Cycle Times" \
  -varname cycles_int_div -value 5.0 -type float
parameter "Cycles Per Load/Store" -group "Cycle Times" -varname cycles_mem \
  -value 2.0 -type float
parameter "Instructions Issued Per Cycle" -varname inst_per_cycle \
  -value 1.0 -type float

###########################################################################
#
# Abstract Commands
#
# command name [-group name] [-method method -filetypes list_of_filetypes]
#   where command is one of
#     view
#     analysis
#     optimization
#     action
#
###########################################################################
view "Execution Time" -group Cost -method execTime -filetypes {nil}
view "Dynamic Instruction Count" -group Cost -method instCount \
   -filetypes {nil}

###########################################################################
#
# Method Definitions
#
# meth name filetype script
#
###########################################################################
#
# execTime --
#
# Determines execution time based on profile from an Application Domain,
# or you can use a default profile (set by the parameters), but that's not
# as interesting.
#
meth execTime  nil {
   if {[lsearch -regexp [context] "Application Areas"] != -1} {
       run set_freq_from_model nil
    }
   lappend freq_time [list "branch" [getParameter freq_branch] \
           [getParameter cycles_branch]]
   lappend freq_time [list "fp_add" [getParameter freq_fp_add] \
           [getParameter cycles_fp_add]]
   lappend freq_time [list "fp_mul" [getParameter freq_fp_mul] \
           [getParameter cycles_fp_mul]]
   lappend freq_time [list "fp_div" [getParameter freq_fp_div] \
           [getParameter cycles_fp_div]]
   lappend freq_time [list "int" [getParameter freq_int] \
                [getParameter cycles_int]]
    lappend freq_time [list "int_mul" [getParameter freq_int_mul] \
              [getParameter cycles_int_mul]]
    lappend freq_time [list "int_div" [getParameter freq_int_div] \
              [getParameter cycles_int_div]]
    lappend freq_time [list "mem" [getParameter freq_mem] \
                [getParameter cycles_mem]]

    set count [getParameter dyn_inst_cnt]
    set clock [getParameter cycle_rate]
    set issues_cycle [getParameter inst_per_cycle]
    set sum 0
    set inst_tot 0

    foreach instruction $freq_time {
      set profile [lindex $instruction 1]
      set inst_tot [expr $inst_tot + $profile]
      set cycles_each [lindex $instruction 2]
      set total_cycles [expr $profile * $cycles_each * $count]
      set cur [expr ($total_cycles/$clock)/$issues_cycle]
      set sum [expr $sum + $cur]
      lappend breakdown "Execution time for [lindex $instruction 0] \
                $cur seconds (profile: $profile, \
                cycles each: $cycles_each, \
                total cycles: $total_cycles)"
     }

    if {($inst_tot <= 0.99) || ($inst_tot >= 1.01)} {
          set RESULT "FAILURE:\nInstruction freq totals not equal to 1.0"
    } else {
          set RESULT [data "Estimated Execution Time" $sum seconds]
          lappend DETAIL [text "BREAKDOWN" [join $breakdown "\n"]]
    }
}
#
# set_freq_from_model --
#
# Gets operation counts from an Application domain and builds a
# profile by adding some estimates for things like branch frequency.
#
meth set_freq_from_model nil {
    #
    # Get the operation counts from the application domain.
    # Assume that the Floating Point Unit will be used if it exists.
    #
    set add 0
    set mul 0
    set div 0
    set fp_add 0
    set fp_mul 0
    set fp_div 0
    if {$hasFlPtUnit} {
        foreach item [run operation_count_model nil] {
           switch -- [lindex $item 0] {
               "real +" { set fp_add [lindex $item 1] }
               "real *" { set fp_mul [lindex $item 1] }
```

```
                "real /" { set fp_div [lindex $item 1] }
            }
        }
    } else {
        foreach item [run operation_count_model nil] {
            switch -- [lindex $item 0] {
                "real +" { set add [lindex $item 1] }
                "real *" { set mul [lindex $item 1] }
                "real /" { set div [lindex $item 1] }
            }
        }
    }
    # estimate we load and store a value 40 times
    set mem [expr 40*[run memory_model nil]]

    # this is core algorithm total operations
    set total [expr $add+$mul+$div+$fp_add+$fp_mul+$fp_div+$mem]

    # floating point intensive and integer intensive programs
     # typically have significantly different profiles
     # (FP: 5% branch, 20% other overhead   INT: 18% branch, 20% other over-
head)
    if {($fp_add+$fp_mul+$fp_div) >= ($total/100)} {
        set total [expr $total + (0.25/0.75)*$total]
        setParameter freq_branch 0.05
        setParameter freq_int [expr ($add / $total) + 0.2]
    } else {
        set total [expr $total + (0.38/0.62)*$total]
        setParameter freq_branch 0.18
        setParameter freq_int [expr ($add / $total) + 0.2]
     }
    setParameter freq_fp_add [expr $fp_add / $total]
    setParameter freq_fp_mul [expr $fp_mul / $total]
    setParameter freq_fp_div [expr $fp_div / $total]
    setParameter freq_int_mul [expr $mul / $total]
    setParameter freq_int_div [expr $div / $total]
    setParameter freq_mem [expr $mem / $total]

    # total count is 5X the calculated count - this is an empirical fudge fac-
tor
    setParameter dyn_inst_cnt [expr int(5*$total)]

    set RESULT "SUCCESS:\n set frequency from algorithm parameters"
  }
  #
  # instCount --
  #
  # Returns an estimate of the dynamic instruction count.
  #
  meth instCount nil {
    run execTime nil
    set RESULT \
        [data "Dynamic instruction count: " [getParameter dyn_inst_cnt]]
  }
}
```

```
#
# DSP56K Architectural Model
#
#
# Author: Ole Bentz, 1996
#
Domain DSP56K -inherit GenericArchitecture {
    ############################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ############################################################################
    key processor dsp56k motorola dsp 56k

    ############################################################################
    #
    # Parameters
    #
    # parameter name [-group menuGroup] -varname varname -value value \
    #  [-type type] [-keep] [-propagate]
    #
    ############################################################################
    parameter "Has Floating Point Unit?" -varname hasFlPtUnit -value 0 -type int
    parameter "Clock Rate (Hertz)" -varname cycle_rate -value 27E6 -type float
    parameter "Cycles Per Branch" -group "Cycle Times" -varname cycles_branch \
      -value 2.0 -type float
    parameter "Cycles Per Int ALU, Logic, Shift" -group "Cycle Times" \
      -varname cycles_int -value 2.0 -type float
    parameter "Cycles Per Int Mul" -group "Cycle Times" \
      -varname cycles_int_mul -value 2.0 -type float
    parameter "Cycles Per Int Div" -group "Cycle Times" \
      -varname cycles_int_div -value 8.0 -type float
    parameter "Cycles Per Load/Store" -group "Cycle Times" -varname cycles_mem \
      -value 8.0 -type float
    parameter "Instructions Issued Per Cycle" -varname inst_per_cycle \
      -value 1.0 -type float
    #
    # Don't have a floating point unit, so set 'em to zero.
    #
    parameter "Cycles Per FP Add" -group "Cycle Times" -varname cycles_fp_add \
      -value 0.0 -type float
    parameter "Cycles Per FP Mul" -group "Cycle Times" -varname cycles_fp_mul \
      -value 0.0 -type float
    parameter "Cycles Per FP Div" -group "Cycle Times" -varname cycles_fp_div \
      -value 0.0 -type float

    ############################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ############################################################################
    #
```

```
# set_freq_from_model --
#
# Gets operation counts from an Application domain and builds a
# profile by adding some estimates for things like branch frequency.
#
meth set_freq_from_model nil {
   #
   # Get the operation counts from the application domain.
   # Assume that the Floating Point Unit will be used if it exists.
   #
   set add 0
   set mul 0
   set div 0
   set fp_add 0
   set fp_mul 0
   set fp_div 0
   if {$hasFlPtUnit} {
       foreach item [run operation_count_model nil] {
          switch -- [lindex $item 0] {
              "real +" { set fp_add [lindex $item 1] }
              "real *" { set fp_mul [lindex $item 1] }
              "real /" { set fp_div [lindex $item 1] }
          }
       }
   } else {
       foreach item [run operation_count_model nil] {
          switch -- [lindex $item 0] {
              "real +" { set add [lindex $item 1] }
              "real *" { set mul [lindex $item 1] }
              "real /" { set div [lindex $item 1] }
          }
       }
   }
   # estimate that load/store of values can be masked by pipelining
   set mem 0

   # Assume that all mults can be done in conjunction with adds,
   # therefore, we just need to find the max of (add,mul).
   # This is often true when processors have a MAC unit.
   set add [expr $add > $mul ? $add : $mul ]
   set mul 0

   # Assume that we always need to load at least one input and store
   # one output. This is true for FIRs and IIRs.
   # Each move costs 3 instructions (the movep is pretty slow)
   # Count it in with the integer (add) operations.
   set add [expr $add + 6]
   set total [expr $add+$mul+$div+$fp_add+$fp_mul+$fp_div+$mem]

   # floating point intensive and integer intensive programs
     # typically have significantly different profiles
     # (FP: 5% branch 20% other overhead  INT: 18% branch 20% other overhead)
   # However, a DSP processor has some special tricks to handle loops
   # etc., so the same profiles don't hold.
   # Instead assume a total overhead of 10%: 5% for branch, 5% for misc.
   # This can cover some branch overhead, some memory overhead, etc.
```

```
        set total [expr $total + (0.1/0.9)*$total]
        setParameter freq_branch 0.05
        setParameter freq_int [expr ($add / $total) + 0.05]

        setParameter freq_fp_add [expr $fp_add / $total]
        setParameter freq_fp_mul [expr $fp_mul / $total]
        setParameter freq_fp_div [expr $fp_div / $total]
        setParameter freq_int_mul [expr $mul / $total]
        setParameter freq_int_div [expr $div / $total]
        setParameter freq_mem [expr $mem / $total]

        setParameter dyn_inst_cnt [expr int(ceil($total))]

        set RESULT "SUCCESS:\n set frequency from algorithm parameters"
        lappend DETAIL [text "ASSUMPTIONS" \
Assuming that one add/sub and one mult can be performed at the same time on \
the MAC unit, thus instead of using number_of_add+number_of_mults we use \
max(number_of_adds,number_of_mults). "\n" \
Assuming that memory access times can be masked by other operations. "\n" \
Assuming that at least one input has to be loaded, and one output \
has to be stored, at a cost of 2 slow instructions (each is 3 times \
slower than an add). "\n" \
Assuming that there is a 10% overhead (e.g., 5% for branches, \
5% miscellaneous) "\n" \
        ]
    }
}
```

```
#
# UltraSPARC Architectural Model
#
# Author: Steven Chan, Nov 1996
#
Domain UltraSparc -inherit GenericArchitecture {
    ############################################################################
    #
    # Keywords for Searching
    #
    # key name [value1] [value2] ...
    #
    ############################################################################
    key processor ultra ultrasparc sun

    ############################################################################
    #
    # Parameters
    #
    # parameter name [-group menuGroup] -varname varname -value value \
    #  [-type type] [-keep] [-propagate]
    #
    ############################################################################
    parameter "Clock Rate (Hertz)" -varname cycle_rate -value 168E6 \
      -type float

    parameter "Cycles Per Branch" -group "Cycle Times" -varname cycles_branch \
      -value 1.0 -type float
    parameter "Cycles Per FP Add" -group "Cycle Times" -varname cycles_fp_add \
      -value 3.0 -type float
    parameter "Cycles Per FP Mul" -group "Cycle Times" -varname cycles_fp_mul \
      -value 3.0 -type float
    parameter "Cycles Per FP Div" -group "Cycle Times" -varname cycles_fp_div \
      -value 12.0 -type float
    parameter "Cycles Per Int ALU, Logic, Shift" -group "Cycle Times" \
      -varname cycles_int -value 1.0 -type float
    parameter "Cycles Per Int Mul" -group "Cycle Times" -varname cycles_int_mul \
      -value 2.0 -type float
    parameter "Cycles Per Int Div" -group "Cycle Times" -varname cycles_int_div \
      -value 4.0 -type float
    parameter "Cycles Per Load/Store" -group "Cycle Times" -varname cycles_mem \
      -value 8.0 -type float

    parameter "Instructions Issued Per Cycle" -varname inst_per_cycle \
      -value 2.0 -type float

    ############################################################################
    #
    # Method Definitions
    #
    # meth name filetype script
    #
    ############################################################################
    #
    # set_freq_from_model --
    #
    # The UltraSparc uses the same model as in the GenericArchitecture domain.
    # It just adds its own parameters as shown above. Therefore, the method is
    # not defined here, since it is inherited from the GenericArchitecture
    # domain.
    #
}
```

APPENDIX E

Database Access Program: Hyperlib

Hyperlib is a program which was developed as an example of an interface that can be used by the Design Server to access a database. Hyperlib gives access to the characterizations of library cells that are available in the Hyper hardware library. Hyperlib has several command line options that specifies which data to pull out of the library. The options are

-s

Show the supply voltage used in the characterizations.

-L lib

Specify which Hyper library to use (default: low_power).

-i inq

Specify what quantity to look for (area,height,width,delay).

-c cell

Specify the cell name to look for (e.g., cla).

-f func

Specify which function to look under (e.g., +).

-p “par value”

Specify a parameter-value pair (e.g., “N 16”).

APPENDIX F

Database Access Program: smv2da

Smv2da is a database access program that provides access to the OCT database used to store the Structure Master Views (SMVs) in the Lager Silicon Compilation System. Smv2da takes as its input an SMV and produces a file with the simple syntax shown below. This file can easily be parsed by a script written in TCL. The information in the file replicates only some of the essential features of the design. The features that are extracted are all the instance names in a hierarchy and all parameters.

The syntax used in the output file consists of a keyword and one or more arguments per line:

```
keyword arg1 [arg2] ...
```

The supported keywords are: designobject, parameter, subcell, and file. The syntax and meaning of each are:

designobject name

Specifies an instance by the name 'name'.

parameter name value

Specifies a parameter 'name' with the given value.

subcell parent_name child_name

Specifies that design object 'child_name' is a subcell of design object 'parent_name'.

file designobject_name filename

Specifies that the database for designobject_name can be found in 'filename'.

A sample output file generated by smv2da is shown below. The design is called "chip," and it has an instance "alu" which in turn instantiates a register (reg), an adder (adder), and two buffers (buf_1 and buf_2).

```
designobject chip
parameter chip LAYOUT_GENERATOR {Flint -bm}
designobject chip_alu
subcell chip chip_alu
file chip_alu {alu/structure_master/contents;}
parameter chip_alu LAYOUT_GENERATOR {Flint -am}
parameter chip_alu STRUCTURE_PROCESSOR {dpp}
parameter chip_alu N {16}
designobject chip_alu_adder
subcell chip_alu chip_alu_adder
parameter chip_alu_adder STRUCTURE_INSTANCE {adder}
parameter chip_alu_adder N {N}
parameter chip_alu_adder CS_TYPE {s}
```

```
file chip_alu_adder {~lager/common/LagerIV/cellib/low_power/dpp/blocks/add/
   structure_master/contents;}
designobject chip_alu_reg
subcell chip_alu chip_alu_reg
parameter chip_alu_reg N {N}
parameter chip_alu_reg CLKINV {0}
parameter chip_alu_reg FB {0}
file chip_alu_reg {~lager/common/LagerIV/cellib/low_power/dpp/blocks/tspcr/
   structure_master/contents;}
designobject chip_alu_buf_1
subcell chip_alu chip_alu_buf_1
parameter chip_alu_buf_1 N {N}
parameter chip_alu_buf_1 SIZE {s}
file chip_alu_buf_1 {~lager/common/LagerIV/cellib/low_power/dpp/blocks/tribuf/
   structure_master/contents;}
designobject chip_alu_buf_2
subcell chip_alu chip_alu_buf_2
parameter chip_alu_buf_2 N {N}
parameter chip_alu_buf_2 SIZE {1}
file chip_alu_buf_2 {~lager/common/LagerIV/cellib/low_power/dpp/blocks/tribuf/
   structure_master/contents;}
```

APPENDIX G

The TILT Language

TILT is the Text-based Interface Language that was developed for the Design Server. It is a textual description of user interface features that can easily be translated to a variety of graphical user interfaces, or to other languages such as HTML. TILT consists of nested lists in the style of TCL, and each TILT component is a TCL list (e.g., {document name body}). TILT lists can be created using standard TCL commands, but it is highly recommended that developers use a set of commands that are provided to build proper TILT lists. The full syntax of each of these commands is given below.

document name item1 [item2] ...

> Documents are generic compound entities that can contain any number of items of different types. Items can be of any TILT type except those that must be in a certain context (i.e., "row" and "col").

text name item1 [item2] ...

Define a block of text with the heading “name”, and with the elements item1, item2, etc. The items can be words, commands, other text blocks, etc.

data name value [units] [type]

Define a piece of data with the name “name”, and with the given value. Optionally, units can be given (e.g., mm^^2), and a type can be specified (e.g., “integer,” “float,” “string”) If a type is not given, it will be inferred.

command name ds_command [arg1] ...

Define a command “name” which invokes the Design Server command “ds_command”. When ds_command is invoked, the arguments (if given) are passed to it.

menu name menu_args

Define a command menu by the given name. The way the name is used is dependent upon translators (e.g., a TILT-HTML translator), but it is typically used as the text displayed on a menu button. The menu_args can be any number of TILT “command” or “menu” items, or an empty list {}. If menu items are used, it will cause hierarchical menus to be created. If an empty list is used, it will create a menu separator, if available in the translator.

selectOne name args

Define a list of options. Only one of the options can be declared "on" at any given time. This can be interpreted as a "radio-button" type interface. Each argument must be a list of {option value [on]} where "option" is the text to display, "value" is the value used if this item is selected, and the optional "on" can be used to specify that an item should be selected.

selectOneMenu name item1 [item2] ...

Define a list of options to be displayed in a menu. One of the options can be selected at a time, and the label of the menu (button) takes on the value of the selected item. If a user interface is not capable of this type of menu button, this keyword should function like selectOne. Each item must be of the form {option [on]}, where option is a string and "on" indicates which of the items is on initially.

selectMultiple name args

Define a list of options, any of which can be selected at a given time. This can be interpreted as a "check-button" type interface. Each argument must be of the form {option value [on]}, and any number of the options can be declared "on."

listSelectOne name size args

Define a (scrollable) list of options of which one can be selected at a time. The "size" argument indicates the visible length of the list. Each argument must be of the form {option value [on]}.

listSelectMultiple name args

Define a (scrollable) list of options of which any number can be selected at a time. Each argument must be of the form {option value [on]}.

table name row1 [row2] ...

Declare a table consisting of TILT rows.

row name col1 [col2] ...

Declare a row (in a table). A row consists of TILT columns.

col name item

Declare a column item (in a row, which is in a table). The item can be: one or more words, a command, selectOne, selectMultiple, entrybox, textbox.

entrybox name [string] [width] [silent]

Define an entry box where the user can enter/edit one line of text. Initialize it with the optional string. Optionally, define the width of the box in characters. The "silent" option can be used to enter passwords or other sensitive information.

textbox name [text] [width] [height]

Define a text box where the user can enter/edit several lines of text. Initialize it with the optional text. Give it the optional width and height if the interface supports it.

submit name ds_command args

Define a "submit" command, i.e., a command which specifies what to do with data which is entered by a user through entryboxes/textboxes/selectOne/etc. The arguments should be any combination of words and names of the entry items that appear in a document. The submit command MUST appear at least one "level" higher than ANY entrybox/textbox/etc. in a TILT document. Here is an example in TCL syntax:

```
set tilt_doc [document "Login" \
       [text "" Please enter your name: [entrybox user "" 20]] \
       [submit OK ds_login -user user]]
```

This example creates a TILT document called "Login", which contains a "text" item and a "submit" item. The "text" item has no heading (""), and contains four words and a TILT entrybox. The "entrybox" has the name "user" (which will reappear in the "submit" item), no initial text (""), and will be 20 characters wide. The "submit" command will appear as "OK" (e.g., on a button), and when invoked the command "ds_login" will be sent to the Design Server with the arguments "-user joe", if the user typed the string "joe" in the entrybox. Notice that the word "-user" passes through untouched, but the word "user", which matches the name of the entrybox, is replaced by the contents of the entrybox. Notice also that the "submit" item is not inside the "text" item (which would put it at the same "level" as the entrybox), but is (at least) one level higher.

tiltError error_message

This routine returns a TILT document, as if this command had been executed:

```
document “Design Server Error” [text “Error Message” $message]]
```

image name type file

Define an image of the given type (e.g., GIF, JPEG) found in the given file. If an image type is not supported by a user interface, the behavior is dependent on the translator.

APPENDIX H

Practical Considerations Regarding the Design Server

Since the Design Server runs as a background process and has no user interface built-in, it is necessary to explain how designers can use external interfaces to interact with the Server. The following sections describe some practical considerations on how to start the Server and interact with it.

H.1. Starting the Design Server

The Design Server requires a certain operating environment which includes paths and environment variables. This environment is defined in a "setup" file which the designer needs to source prior to starting the Design Server.

One of the environment variables that are defined in the setup file is called "DA_PORT". This variable is used to determine the port number which the Design Server will use when setting up its communications port. The value of "DA_PORT" is important because it is a part of the address which external interfaces need to connect to the Server.

The full address, following the conventions used in the World Wide Web, is

```
http://hostname:port/designAgent
```

where hostname is the name of the computer where the Server is running, and where port is the value of the DA_PORT environment variable.

After sourcing the setup file, the Design Server can be started from a UNIX prompt with this command:

```
% da &
```

This command starts the Design Server as a background process. The Server loads all known domains, and opens the communication port described above. It then waits for a designer to initiate access through an external user interface or a design interface tool.

The Design Server occasionally needs to print run-time information on the terminal from which it was started. However, this information can be redirected to a designer's user interface, to enable the designer to view it from any location.

The Design Server can alternately be started from a Web browser. In this case most of the issues discussed above can be hidden or automated, but the Server will run as a generic user without the privileges the designer may otherwise have. There has to be a set of underlying scripts or programs to automate the process of setting up the proper environment and invoking the Design Server. One implementation has been developed, which dispatches the Server to a remote host, automatically determines an available port number, sources the setup script, and provides convenient links that allow the user to point-and-click to access the Server.

H.2. Modes of Interactions

Interactions take place through user interfaces that are built on top of the text-based language "TILT" (see Appendix G), which was developed specifically for the Design Server. User interfaces are external clients that access the Design Server through the network. There are two useful forms of interfaces; graphical user interfaces and text-based interfaces. In most cases a graphical user interface can better convey information to the user, and at the same time can facilitate text-based input. The only user interface that has been fully developed translates the Server's TILT output to HTML, which can be viewed in a Web browser such as Netscape Navigator and Microsoft Explorer. In these cases the Web browsers are essentially used as graphical user interfaces. The main requirements for graphical user interfaces are that all TILT elements can be represented graphically, and that the interface is capable of accessing the Design Server via the Web.

Interfaces can also be purely textual, which is convenient for certain interactions, such as a "single request" interaction. Single request interaction is when design specifications and a request are given together, and the search engine is used to propose appropriate contexts before the request is interpreted and evaluated. Such interface should be available as a regular command (e.g., from a UNIX command line), and can be relatively simple, providing just the following functionality:

1. Facilitate User Identification - prompt the designer for user name (or find it automatically, if possible) and password.

2. Accept Specifications - prompt the designer for a list of design specifications for use by the search engine. One of the specification words should be a request for information.

3. List Proposed Contexts - show the list of contexts proposed by the search engine, and let the designer choose. Alternately, if desired, automatically choose the first context.

4. Facilitate Command Resolution - after the context is chosen, the environment can interpret the request. If there are several possible interpretations, the designer must be prompted for a decision. Alternately, if desired, the first command can be chosen.

5. Show Results - display the results that are returned from the environment.

As a third alternative, the Design Server can be encapsulated by other tools. One example of such a tool is the High Level Block Editor (HLBE) developed by Shola Olateju [Olateju96]. The HLBE lets a designer construct a high-level block diagram of a design, and each block can be annotated with properties. The HLBE connects to the Design Server, sends the block diagram, and conveniently opens a Web browser to let the designer interact with the Server. The Server tries to establish a context for each block (object) by using the given properties to find relevant domains.

H.3. The Design Object Interface

The Design Server is an object-oriented environment, so it is important to describe the designer's view of the main object, the design object. Almost all commands have to be directed to a design object. The exceptions to this are the "global commands" which allows designers to choose between existing design objects, and to shut the Server down.

A design object is capable of listing the requests that can be handled in the current context. This is an important capability since the set of possible requests changes over time as the object's context or design specifications change. Figure H-1 shows the list of available requests, in groups or lists called "menus," and it also shows the result of making a request, whether the request is going to return another list of commands (e.g., the administrative request), a list of names and values (e.g., the parameters request), or perhaps a prompt for more information (e.g., the specification request).

The design object is always able to handle a set of "static" commands to view or modify the context, the parameters, etc. It also has four dynamically changing categories, View, Guidance, Optimization, and Action. Each of these categories will show only the commands that are available and "enabled" in the given context. A command is enabled when the context contains at least one script which can perform the command, and the design object has at least one file of the types required (if any) by the available scripts.

A few items deserve explanation:

Details Menu - lists the available details associated with previous results.

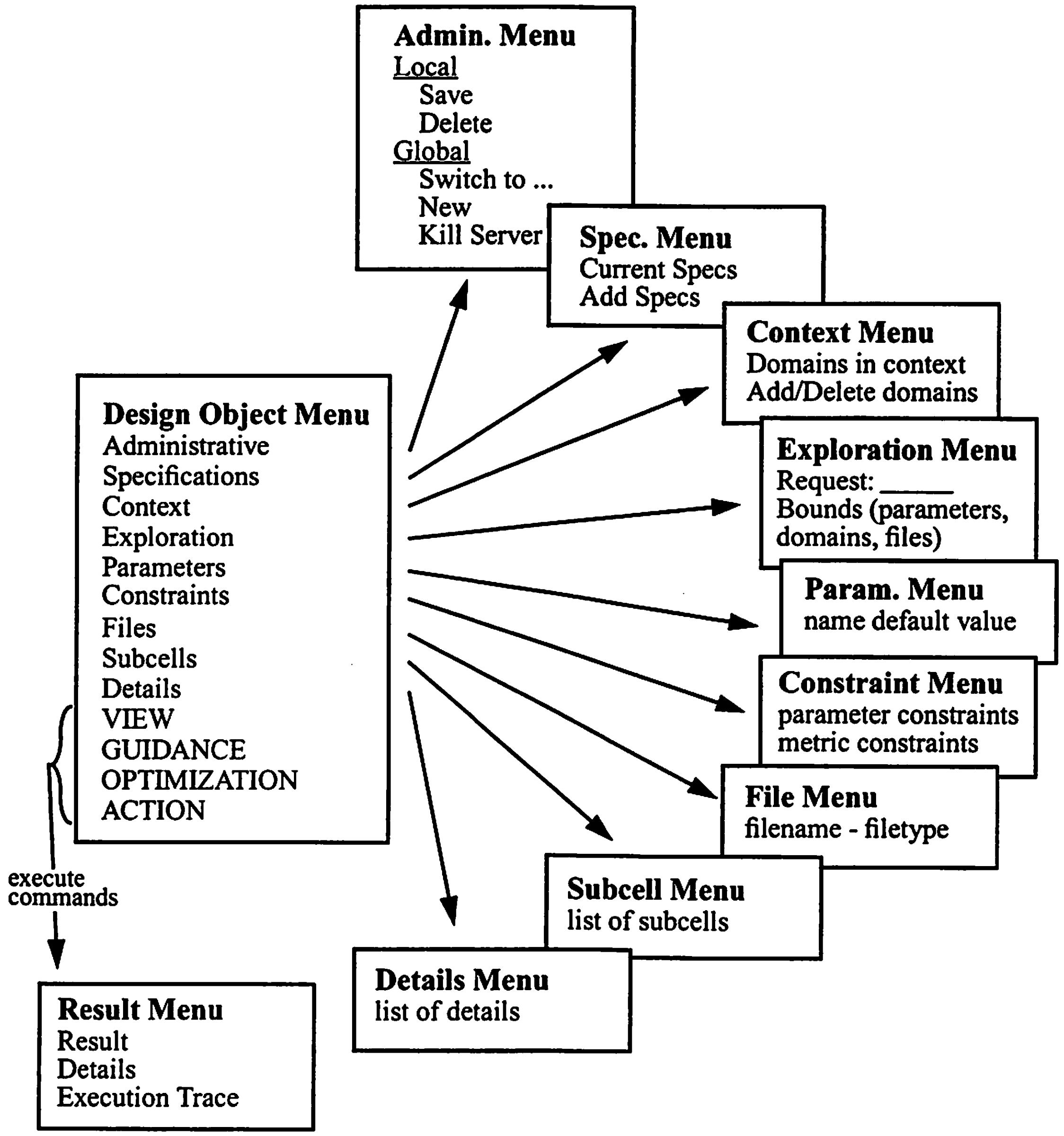


Figure H-1. Menu Organization.

Exploration Menu - facilitates automated exploration, where the designer has to give a request and the bounds of what to explore. Exploration can be based on parameters, a list of domains, or a list of files. For parameters, the starting value, increment and ending value have to be given. For domains, a list has to be given, and each domain will in turn be substituted into the context, before the request is evaluated. For a list of files, the request will be evaluated for each of the given files.

Files - shows the files that are associated with the design object. It also provides a way to add or remove files from the object. Usually file management is handled transparently, but in some cases it is necessary to notify a design object that a certain file should be considered a part of the design.

Result Menu - shows the result obtained by evaluating a script. It can also list any number of "Details." Details are created by a script to communicate information in addition to the result. A typical example of a detail is to show the breakdown of a summation, such as how the components of a chip contribute to the overall area. The result menu also has an "Execution Trace" command, which is generated automatically as a script is evaluated. This is useful after-the-fact to determine how a result was produced. Since scripts often call other scripts, and since it is sometimes determined at run-time which script will be called, the execution trace can be very informative.

H.4. Summary

This appendix gave an overview of some practical considerations regarding the use of the Design Server. This information should be adequate to help a new user get started. For an overview of typical design flows, please read Chapter 6.