This is to certify that the

dissertation entitled

Configuration Management and Version Data Modeling in VLSI Design Environments

presented by

Sangchul Kim

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date March 2, 1994

Configuration Management and Version Data Modeling in VLSI Design Environments

By

Sangchul Kim

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DEGREE OF PHILOSOHPY

Department of Computer Science

1994

ABSTRACT

Configuration Management and Version Data Modeling in VLSI Design Environments

By

Sangchul Kim

This thesis presents techniques for version data modeling and configuration management in VLSI design environments. These techniques are based on an object-oriented data model, which efficiently captures diverse semantics of VLSI designs and the design process. Relationships between versions are identified, and formally defined using design constraints. Constraints are also utilized for increasing designers' control over configuration binding, so designers can ensure the correctness of configurations or select components of preferred design styles. An algorithm is presented which minimizes the total area of a design, resulting from configuration binding under a maximum allowable delay. This algorithm achieves significantly better performance than previous work. A workspace model is presented which hierarchically organizes workspaces, configurations of CAD task-related design objects. Mechanisms for the model provide the snapshot and inheritance property, which facilitate system support for change propagation, release control, and design tracking.

To my parents and my family

ACKNOWLEDGMENTS

I wish to express my gratitude and appreciation to my thesis advisor Dr. Moon Jung Chung for his consistent guidance and encouragement right from the beginning, and his financial support. I am grateful for many discussions and invaluable comments he provided.

I would like to thank my guidance committee members, Dr. Anthony S. Wojcik, Dr. Lionel M. Ni, Dr. Betty H. C. Cheng, and Dr. Raoul Lepage, for their help and guidance.

I also thank my wife Hosuk Hoang and my two daughters, Jungmin and Jungju, for their patience and love during many long days. I must express my heartful thanks to my parents for their sincere prayer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Problem Definition and Our Approach

The design of a VLSI system is a complex activity, which involves numerous types of information and enormous amounts of data. A system-level design is usually broken down into smaller parts, and a design team cooperates on those parts. Since this design style is, by its nature, iterative and tentative, different versions of parts of a design exist over time. An entire design is built from parts developed by different team members. Designers working on different parts keep track of an entire design, coordinate updates to the design, and ensure data consistencies among them. With Computer-Aided Design (CAD) tools, presently designers synthesize a large, complex circuit from a user-specified description, for example, a VHDL behavioral model. Designers explore various design alternatives until they obtain a satisfactory output. Design alternatives result from the choice of values for the design parameters, CAD tools for the same CAD function, etc. Therefore, designers manage versions of designs produced from design alternatives, and organize them into a design history.

In VLSI/CAD frameworks, design data management has primarily concentrated on version control and configuration management [9, 10, 21, 22, 46, 60, 81]. Version control is a collection of services for tracing the version history and structuring rela-

tionships, such as revisions, between versions [10, 22, 47, 65, 73]. In CAD frameworks [7, 22, 50] and software engineering [27, 33, 68, 73], a *configuration* is a collection of related, versioned components and it is a unit of access. Configuration management is system support for configuring a large object (*i.e.*, a configuration) from pre-designed components (called *configuration binding*), preserving states of components of an entire design, and keeping data consistencies among designers.

Based on types of components, configurations can be classified into the following three levels: *chip level*, *process level*, and *development level* (or life-cycle level). Chip-level configurations are large designs built around smaller designs. Chip-level configuration binding can be aimed at achieving two challenges. One is to make CAD frameworks guide designers in configuration binding with respect to component selection and constraint checking. The other is to make sure that configured designs minimize a performance measure, such as delay or area, under limitations on other measures. CAD tasks are carried out by specifying design processes and executing them [35]. A design process is defined as a combination of tool invocations and/or subprocesses that perform a CAD task [35]. Process-related data (data produced or read during a design process) needs to be placed in areas controlled by data managers. Process-level configurations denote those areas, called *workspaces* in the literature [7, 27, 68, 77].

Development-level configurations are collections of data which are handled through various stages of the whole life cycle, such as testing, documentation, and implementation. This level of configuration management is primarily studied in software engineering [56].

This thesis presents the following:

- an object-oriented VLSI data model,
- a constraint-driven methodology for version modeling and chip-level configura-

tion binding,

- an area-optimization algorithm for chip-level configuration binding
- a method of workspace management for design processes.

Since the relational data model is not sufficient for representing the rich semantics of complex VLSI systems, many researchers have investigated object-oriented VLSI data modeling [3, 9, 39, 81]. However, few previous object-oriented VLSI data models can sufficiently capture *abstraction* relationships between design objects. Such relationships exist between design objects which result from the stepwise refinement nature of VLSI design. In our object-oriented data model, design objects are hierarchically organized based both on abstraction relationships between them and on design constraints enforced on them. This thesis will show how our data model can also provide a basis for capturing a rich set of version relationships. Little work, except for keeping versions of configurations, has been reported on system support for chip-level configuration binding. In our configuration binding methodology, user-specified design constraints are utilized to detect design inconsistencies, realize certain design styles preferred by designers, and enforce user-specified values of design parameters. The data model and the configuration binding methodology presented herein were originally developed for VHDL [42] environments. However, we claim that basic underlying ideas are general enough to be applicable to other VLSI/CAD design environments.

One algorithmic problem in chip-level configuration binding [16, 23, 58] is finding an area-optimal list (called *a binding*) of technology-specific cells for a circuit with logic gate-level components. Several algorithms [16, 58] have been suggested which iteratively choose several components and find a new binding for them. Our algorithm for this problem iteratively selects *paths* of gates and finds a solution, and then uses a cloning method to improve the solution further. In this cloning method, a series-

parallel graph [75] is obtained from a circuit by duplicating nodes (components) into clones. Together with the cloning method, the path-oriented nature of our algorithm provides significantly better solutions than previous work.

In the CAD framework literature [7, 10, 22, 60, 77], few workspace models have been shown which can support team design efficiently. The workspace model presented herein can capture the way in which the entire task of a project is partitioned into subtasks. In our workspace model, a workspace is created for each subtask, and workspaces are organized hierarchically. The content of a workspace is visible to all of the designers that work on its descendant workspaces. A mechanism is presented which preserves design states of a project. This mechanism and the workspace hierarchy will be shown to enable easy control over design tracking, change propagation, and release control.

To understand how our techniques for version data modeling and configuration management fit into CAD frameworks, let us consider a conceptual CAD framework illustrated in Figure 1.1*. In the framework, those techniques are embodied into procedural interfaces to the data store. Designs are archived in the design library, located at the bottom of the architecture. Workspaces contain designs which designers work on. After designs are modified in a workspace, they are archived back into the design library as new versions. Access to designs and workspaces is through three procedural interfaces. One procedural interface, named "basic access/version control PI", is a collection of basic routines for accessing (reading, creating, or deleting) designs in the design library and workspaces, and performing version control over designs. Two other procedural interfaces, named "chip-level CM PI" and "process-level CM PI", are collections of routines for chip-level configuration management and workspace management, respectively. CAD tools can be built on top of these three interfaces.

*In the figure, life-cycle level configuration management is not considered, as in almost all of existing CAD frameworks. Two terms, CM and PI, stand for configuration management and procedural interface, respectively.

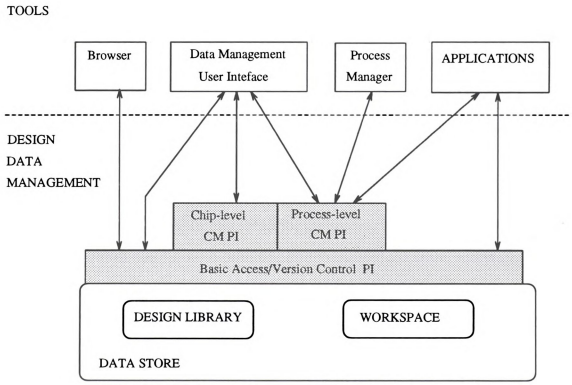


Figure 1.1. The Architecture of a Conceptual CAD Framework

Among them, a browser is a tool essentially needed for design data management, which allows designers to navigate through the design library. A CAD tool named "data management user interface" provides a user interface through which designers can explicitly execute data management operations in the three procedural interfaces. A scenario of how the framework is typically used is as follows. First, a designer interacts with the browser and selects a set of objects to work on. Next, the selected objects are checked out into a workspace through the user interface. The designer carries out a CAD task in his workspace by executing CAD tools, such as an editor or a tool for configuration binding, on checked-out objects. Finally, outputs of tools (objects that are produced from other objects by CAD tools) are checked into the design library through the user interface. A variation to the scenario is as follows. Instead of executing CAD tools manually, the designer runs a process manager, a CAD tool that executes a sequence of tools for a CAD task [35], on checked-out objects in his/her workspace. In this scenario, the process manager reads objects in the workspace through the "process-level CM PI", invokes CAD tools, and stores outputs of tools in the workspace through the interface. The process manager can create new workspaces for subtasks of its target CAD task through the "process-level CM PI".

1.2 Previous Work

1.2.1 Electronic Design Frameworks

The techniques presented in the thesis are related to three fields in electronic CAD frameworks: *object-oriented VLSI/CAD version data modeling*, *configuration binding*, and *data management for design process execution.*

Since record-based data models do not directly represent the complex semantics of VLSI circuits, the object-oriented paradigm has been exploited in VLSI/CAD data modeling. Batory and Kim's work [9] can be regarded as the first coherent

object-oriented data model for organizing VLSI design data and bridging the concept of version with other structural concepts. Their model separates a design, called a molecular object [9], into two levels: an interface (port), and an implementation description. Versions are defined to be various implementations of an interface. One limitation of the model is that it forces all implementations of an interface to be placed at the same level in the design library. Therefore, it cannot distinguish designs in terms of architecture, technology, constraints, etc. Several approaches have been proposed that incorporate more abundant concepts in object-oriented data models, rather than simply identifying design data as objects. They are: 3DIS [3], DMS [81] and CBASE [39]. 3DIS, adopted in the ADAM system [38], represents multiple views of designs, such as structural, behavioral, and timing, in a unified data format. DMS, used in the ICD-NELSIS system developed by Delft University, formally integrates the high order concepts at the object type level, such as equivalence, composition, and version, into a single data schema. This integration enables us to unambiguously recognize all relationships between object types. CBASE has a repository of type (class) definitions for circuit elements. CBASE models ports and wires as objects. While 3DIS, DMS and CBASE have increased the data modeling power of the object-oriented paradigm, they, in common, do not distinguish alternatives and versions. None of the models separate interfaces and implementations, thus making it difficult to relate multiple implementations to an interface.

Fred [80] provides a procedural language, Ethel, to represent structural and layout objects and measurement functions. Ethel has predefined four classes for components, pins, wires, and wire segments. It supports features of object-oriented modeling, which include subclassing of the four classes, data typing, and inheritance. Ethel does not enforce one fixed data model, but enables us to build an arbitrary model using its language constructs. W. H. Wolf claim that this generality is flexible but inefficient for data management [80]. Additionally, Fred does not take version control

into special consideration. The data model of the PLAYOUT system [71] is similar to the data model presented in Chapter 2. The model separates the entire data of a design entity into interface, architecture and configuration. However, the PLAYOUT system does not treat constraints as one of the directions along which designs evolve. As a result, it provides mechanisms neither for controlling configuration binding nor for clustering design alternatives based on design constraints. Moreover, the data model of the PLAYOUT system lacks important concepts, such as incompletely bound configurations, which can serve as a useful vehicle in representing the stepwise refinement of configuration binding [20, 21].

Previous version models [10, 47, 60, 81] are almost equivalent to each other in the sense that they organize designs based on three basic relationships: revision, equivalence, and configuration. Among the models, the VERSION SERVER system [47] introduces various high-level relationships, such as configurations of equivalences and versions of equivalences. It also has several innovative concepts, such as layers of versions and dynamic configuration binding. However, since the data model of the VERSION SERVER system treats entire designs as modeling units, it does not distinguish interfaces and implementations. As a result, it does not allow revisions and alternatives of designs at the interface or implementation level.

There are three levels of configuration management – chip level, process level, and life-cycle level, as mentioned earlier. Life-cycle level configuration management is not a focus of this thesis, and little work has been published on it. The major task of chip-level configuration management is configuration binding. Chip-level configuration binding can apply to two stages of VLSI design: when a designer is building up a large circuit from predesigned components; when a designer is targeting a technology-independent, structural object to a technology by binding each generic component to a technology-specific cell. Few systems, except for the CADENCE DMS [60], have a configuration binding service for the first stage (*i.e.* building of a large circuit).

However, configuration binding of the CADENCE DMS is extremely limited in that a designer is forced to follow fixed dynamic binding schemes, and cannot be assisted in error detection.

In the logic synthesis literature, a few systems, such as LORES-2 [32], POLARIS [69], and Socrates [37], have been developed which can target a technology-independent design to a technology. These systems accept a behavioral description or an equation, transform it into a technology-independent logic design, and finally produce a technology-specific structure. Since all of those systems are targeting a logic level design, they are not suitable for system-level designs built around larger components than gates, for example, RTL-level components. Few systems, except for VSS [59], can target system-level, technology-independent designs to a technology through configuration binding. VSS is a high-level synthesis system that accepts a VHDL description and generates a schematic of technology-independent system-level components. The schematic is later fed to MILO [76] which obtains a technology-specific binding and generates a layout. Since VSS does not have a underlying data model for organizing design data, the representation and retrieval of designs are inefficient. Moreover, another limitation of VSS is that a designer cannot guide configuration binding with user-specified constraints.

Process-level configuration management is workspace management for design processes. A large number of VLSI/CAD frameworks, notably Ulysses [13, 14], Cadweld [30], NELSIS-CAD [12], Sigma CAD framework [40], the Tzi-cker's system [18], Minerva [43], and VOV[15], have been developed which support process control. They are primarily aimed at executing a user-specified description of a CAD task by invoking CAD tools. However, their capability of data management for process executions is limited. NELSIS-CAD provides only one option that selects whether an old version of a tool output is kept or overwritten by a new version if the same tool is invoked again. The Sigma CAD framework has the notion of data set, which is a complete set

of files generated by a single run of a design process. If a large task involves more than one process, keeping data sets of each process separate is not sufficient for preserving design states of the whole task. VOV and the Tzi-cker's system provide a capability that automatically records a history of tool invocations and data dependencies. The capability is useful for a designer who explores various design alternatives. However, they enforce all design alternatives of a process to be explored in one workspace, and provide little system support for identifying how explored design alternatives differ from each other. Few CAD frameworks, except for the DDM system [7], use workspaces as a basis for process-level configuration management. The DDM system hierarchically organizes workspaces for allowing designers to easily create a workspace identical to another except for minor variations. However, it does not offer a discipline as to how the entire task of a project should be distributed across workspaces. Moreover, it lacks control over newly released versions, which makes it difficult for designers to coordinate changes in the context of a project.

1.2.2 CASE (Computer-Aided Software Engineering) Systems

Version control and configuration management are active research topics of CASE (Computer-Aided Software Engineering). The aim of version control and configuration management in CASE is to help designers manage the evolution of designs (*i.e.*, source code and object code), texts, and various products generated during the software life cycle (*i.e.*, manuals, specifications, etc). Instead, version control and configuration management in VLSI/CAD has focused only on designs, which are VLSI circuits.

There have been a large number of CASE systems for version control and configuration management, notably RCS [73], SCCS [66], DSEE [55], Inscape [65], Cedar's

System Modeler [54], Gypse [27], NSE [2], the NuMIL system [63], Winkler's work [79], Make [34], and Adele [33]. Most of the CASE systems, such as RCS, SCCS, DSEE, Gypse, and NSE, trace changes of files in version control. Since units of access are designs (hardware circuits) in VLSI design, those systems are not adequate for VLSI/CAD version control.

Inscape, Modeler, and the NuMIL system offer module-level, not file-level, version control and configuration management, where a configuration is defined as a composition (combination) of software modules. Inscape includes a module interface specification language, called Instress, to describe the information about modules, such as the property of arguments and the behavior of operations. The NuMIL system has a module interconnection language, called NuMIL, which is functionally similar to Instress. Inscape and the NuMIL system can analyze the information about modules, and check the consistency and completeness of a configuration. However, Instress and NuMIL originally were designed for software module specification. Hence, their language constructs, such as constructs for specifying conditions on buffer allocation and file opening, are not adequate for capturing semantics of VLSI systems. Modeler provides Cedar-language programmers with version control and configuration management during the development cycle. The data model underlying Modeler is too restrictive, since versions of a modules are identified by unique names and their creation dates, and only one line of versioning for a given module is allowed. Adele utilizes attributes associated with versions of modules, and combines only versions satisfying constraints into a configuration. Since configurations in Adele are sets of software modules, constraints to be enforced for configuration binding of hardware circuits, such as an upper bound on the total area of selected modules, cannot be specified.

Process-level configuration management is provided by Make [34] and Make-alikes [6, 26]. These systems keep track of changes to program files and rebuild a

system (configuration) once such changes have taken place. Especially, Amoeba [6], Odin [26], and BiiN [68] automatically find build steps from relationships between files, such as inclusion and external procedure calls, and file types. The systems have no notion of version; the consistency, ambiguity, and completeness of a a configuration are left for users to resolve. Moreover, they were developed as a single-user environment, so they do not provide services for processes that are collaboratively performed by more than one designer.

Few CASE systems, except for Shape [61] and Gypsy, have introduced the concept of a workspace as a basis for team design. The notion of a workspace in Shape is developed to isolate a user from interfering with others' work. Gypsy enhances the concept such that a workspace keeps only versions of interest to a user, while other versions are invisible. The notion of a workspace in the two systems is simple and may be applicable to a small design team. The systems do not consider how workspaces for a project should be organized to support important services for team design, such as change propagation and release control.

1.2.3 VHDL Design Environments

Since the version data modeling and configuration binding presented herein are aimed at VHDL design environments, let us describe version control and configuration management of previous VHDL design environments. To date, various VHDL design environments, notably those of IBM [67], MCC [1], Mentor Graphics [28], CAD Language System, Inc. [62] and Intermetrics, have appeared. Those systems integrate various point tools, including simulators, schematic editors, and high level synthesis tools, by using a common place for data storage. However, in each of these systems, its data model is simply a directory/file structure in which the outputs of tools can be placed. Therefore, the data models of the systems are simply determined from a collection of point tools are integrated into the systems.

For VHDL users, configuration binding is the final step to complete a design. None of previously developed VHDL environments offer system support for combining existing components into a configuration. VHDL design environments can inherently support the methodology for technology-independent design as follows. Designers separate technology data from VHDL architectures, and represent it in VHDL packages and VHDL configurations. Then VHDL architectures are technology-independent. Configuration binding on them can be performed later to map them to a technology. Previous VHDL environments lack many useful concepts which facilitate configuration binding for technology mapping. For example, none provides stepwise binding and area (or delay) optimization.

1.3 Overview of the Thesis

The structure of this thesis is as follows. Chapter 2 presents an object-oriented VLSI data model, on which our methodology for version modeling and configuration binding is based. Chapter 3 presents a constraint-driven methodology for version modeling and chip-level configuration binding. In Chapter 4, an algorithm for chip-level configuration binding is presented which finds a binding of the circuit with gates under a maximum allowable delay. Chapter 5 describes a methodology for workspace management in design processes. Chapter 6 concludes with a list of further research issues.

CHAPTER 2

Data Modeling of VLSI Design Data

An object-oriented VLSI data model is presented that supports multiple levels of abstraction, multiple domains, and versioning. This model is originally developed as a framework for the logical organization of design data in VHDL design environments. Our data model is characterized by an abstraction hierarchy of design objects, which has two major features: each object can be seen as an abstraction of all its descendant objects; constraints are assigned to design objects and satisfied by their descendants. This abstraction hierarchy makes it easy to represent the multi-dimensional evolution of design objects in a seamless manner, locate a design object with certain properties, and efficiently support data management mechanisms, such as version control and configuration management. Our data model enables a technology-independent design methodology, in which only configurations encapsulate technology-specific data and other design objects are left technology-independent.

2.1 Introduction

The designing of VLSI circuits is a complex process that produces design data in multiple domains such as behavioral and structural. The design process evolves in multiple dimensions: refinement, composition, revision, and transformation. VLSI design environments should capture the process of transformations and the relationships between designs at various levels of abstraction and in various domains. They must provide mechanisms for design library, version control, and configuration management. The modeling power of a data model determines how conveniently these mechanisms can be supported.

Since conventional (relational) data model cannot efficiently capture the semantics of the object, the object-oriented paradigm has recently been employed for modeling VLSI objects [3, 8, 9, 21, 38, 39, 47, 80, 81]. The important features supported by the object-oriented paradigm, such as aggregation, classification, and inheritance, are used for exploring semantic data models for VLSI objects [9, 81], to simplify the implementation of design utilities [39], and to compute performance measures [80] hierarchically. However, the previous data models have limited capabilities of representing various semantics of designs that evolve in various dimensions and multiple levels of abstraction.

This chapter presents an object-oriented data model.* The proposed data model can capture design evolution in a seamless way, and support a set of modeling concepts necessary for a constraint-driven approach to version control and configuration

*Our data model is object-oriented from Ullman's definition of "object-oriented" [74]. In [74], Ullman says that a system is object-oriented if its capabilities of data modeling include three features: complex object, encapsulation, and object identity. A complex object denotes an object that is an aggregation of constituent objects. Encapsulation is the ability to define procedures applying only to objects of a particular type and the ability to require that all access to those objects is via application of one of these procedures. An object identity denotes a handle which identifies an object from others.

management, presented in Chapter 3. A basic principle of our data model is that various abstraction levels of VLSI systems, attributes, and design constraints are modeled as objects [24], which are the units of access and retrieval. A design entity is an integral object whose entire data is separated into four levels of abstraction: functionality (operation type), interface (port description), generic implementation, and configuration. The projection of a design entity on each level of abstraction is modeled as an object, called a *design object.* A "generic implementation," composed of *generic components*[†] and interconnections, describes an architecture of the design entity. A "configuration" is derived from an implementation by *configuration binding*, a process of selecting specific design objects for generic components. That is, a configuration represents the entire data of a design entity. Our concept of "four levels of abstraction" is uniformly applicable to each of various domains. For each domain, design objects form an abstraction hierarchy according to their abstraction levels.

Another feature of our data model is that design constraints can be associated with design objects at any abstraction level, and enforced on their descendants as well as the objects. Constraints serve as a vehicle to validate the correctness of design objects. Configurations can be used to encapsulate technology-specific data, allowing design object at all other levels to remain technology-independent. Then, a technology-specific design object is obtained by binding all components of an implementation to technology-specific design objects. The appearance of a new technology may introduce changes in the design library, thus it requires earlier VLSI systems to be redesigned. Our technology encapsulation minimizes this problem, since only a new configuration is needed for retargeting an existing technology-independent design object to a new technology. Except for the PLAYOUT system [71], there is no previ-

[†]A component is *generic* if it can be elaborated into a specific object later. For example, consider a component X which is an interface object of a design entity O. If the entity O has several implementations for the interface, the component can be elaborated later by binding X to one of the implementations.

ous data model that was designed to organize *system-level* VLSI systems adequately for this design style.

The data model proposed herein was originally developed as a basis for organizing design objects in a VHDL [42] design environment called *SDE* [21, 22]. It is assumed that all design objects in behavioral and structural domain, handled in the data model, are written in VHDL. Design objects in other domains are also allowed, which can be written in proper languages, such as EDIF for the layout domain.

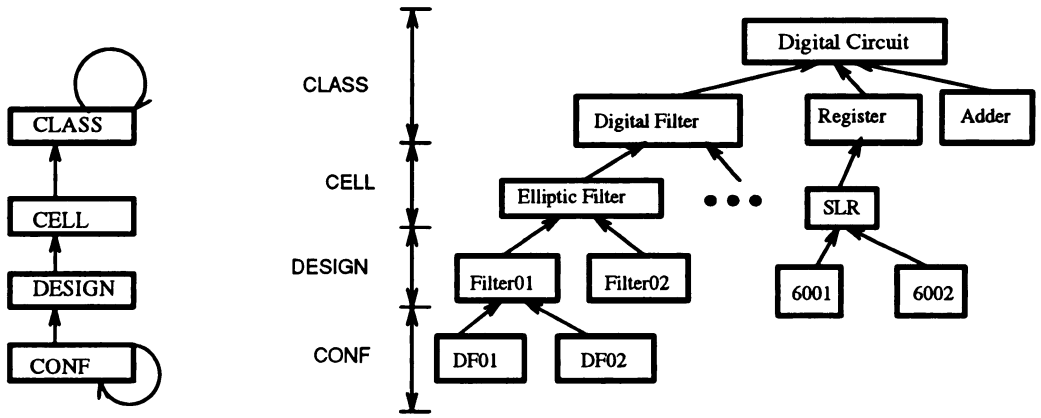
The rest of the chapter is structured as follows. Section 2.2 describes an object-oriented data model for designs in the structural, behavioral, and layout domains. Section 2.3 describes a data schema of the proposed data model. In Section 2.4, we briefly highlight advantages of the data model in the development of VLSI/CAD applications. Section 2.5 concludes.

2.2 A Data Model for VHDL Design Objects

Our data model has six types of VLSI objects: design entity, port, wire, attribute, constraint, and package. *Attributes* represent different information characteristics of designs. They are similar to VHDL attributes except that their values can be *functions* as well as literals. A *constraint* is a special kind of attribute that is evaluated to a boolean value *true* or *false*. It is assigned to design objects, and later serves as the criteria of verifying their correctness. A *package* represents a VHDL package that is a set of definitions for user-defined symbols, such as function names, variable types, and symbolic constants [42].

2.2.1 Four Levels of Abstraction

Our data model captures the data of design entities at multiple levels of abstraction. Figure 2.1 (a) shows four different levels of abstraction: CLASS, CELL, DESIGN, and CONF. Those four levels represent the functionality, interface, generic implementation, and implementation of a design entity, respectively. Objects representing those levels of design entities are called *design objects*. Design objects are hierarchically organized based on abstraction between them. In other words, a design object at a level is an abstraction of all its child objects in the abstraction hierarchy[‡] This abstraction hierarchy can be applied to each of the three domains: *behavioral, structural*, and *layout*. First, we describe the four levels in terms of the structural domain only, and then we will generalize them to the other two domains in Section 2.2.2.



(a) Four levels of Abstraction (b) An Example Cell Database

Figure 2.1. Cell Abstraction Hierarchy

[‡]In [78], the interface for a VLSI circuit is modeled as an abstraction of its implementation.

- A CLASS-level object (in short, a CLASS object) represents a hardware function type (such as addition or division). This level is used to group design objects with similar functionalities and then to facilitate browsing and locating design objects. A CLASS object does not have a corresponding concept in VHDL. This object cannot be a component of another design because it does not have an interface definition.

- A CELL-level object (for short, a CELL object) is an *interface*, which is the description of ports and parameters. CELL objects can be characterized by constraints that are exported down to all their descendant DESIGN objects in the abstraction hierarchy. As a result, these DESIGN objects must satisfy the inherited constraints. A CELL object corresponds to a VHDL entity, but is different in that it can be associated with general constraints, which all of the implementations derived from it must satisfy.

- A DESIGN-level object (for short, a DESIGN object), is a *generic implementation* of a CELL object. Such an implementation is a schematic built around interface objects (*i.e.*, CELL objects)[§] In this respect, a DESIGN object is similar to a VHDL architecture description. In addition, designers can enforce further constraints on a DESIGN object. For example, we can specify for a DESIGN object that its components must be design objects at certain locations of the abstraction hierarchy or that its components must satisfy certain performance constraints. Those constraints are checked to detect the errors of a DESIGN object, or used later for deriving correct descendant objects (CONF objects).

[§]Some DESIGN-level objects in the structural domain, for example a two-input AND gate, are so primitive that they may not be decomposed into smaller objects. Such DESIGN objects have empty structural descriptions, but usually have their corresponding objects in other domains. For example, for an AND gate, a behavioral object exists for its simulation, and a layout object exists for its geometrical information.

- A CONF-level object (for short, a CONF object) is a *configuration* of a DESIGN object. That is, CONF objects represent implementations of design entities. A configuration is generated from a DESIGN object in such a way that its components are bound to (in other words, *elaborated into*) other design objects, which are DESIGN-level and CONF-level objects. This generation process is called *configuration binding.* Depending on selections of design objects for components, various CONF objects are generated from the same DESIGN object. A CONF object corresponds to a VHDL configuration. Constraints can be imposed to restrict a set of design objects selectable for components. They include not only the constraints directly assigned to a CONF object under construction but also those assigned to the parent (DESIGN object) of the CONF object. Note that a DESIGN object can have only one CONF object if the constraints imposed on its component bindings restrict that only one design object can be selected for each component.

Put more precisely, the components of a design object are *references* to other design objects. In our data model, *generic components* are references to design objects that can be elaborated into lower-level objects. For example, CELL objects are used as generic components of a large design since they can be elaborated into their descendant objects, *i.e.*, DESIGN-level or CONF-level objects. Similarly, DESIGN objects are also used as generic components. We allow CONF objects such that some of their components are still generic. Such objects are called *incompletely bound* CONF objects. An incompletely bound CONF object derives other CONF objects by elaborating its generic components. This is the reason that CONF objects can span several layers in the abstraction hierarchy, as illustrated in Figure 2.1 (a). Incompletely bound objects play an important role in version management.

Figure 2.1 (b) illustrates an example of the design library. Figure 2.2 illustrates that a DESIGN object, `Filter01`, is built around two components, `Carry Lookahead` (`CLA`) and `Shift_Load Register` (`SLR`), which are CELL objects. The DESIGN object `Filter01` derives two CONF objects, `DF01` and `DF02`, by binding `SLR` to `6001` and `6002`, respectively.

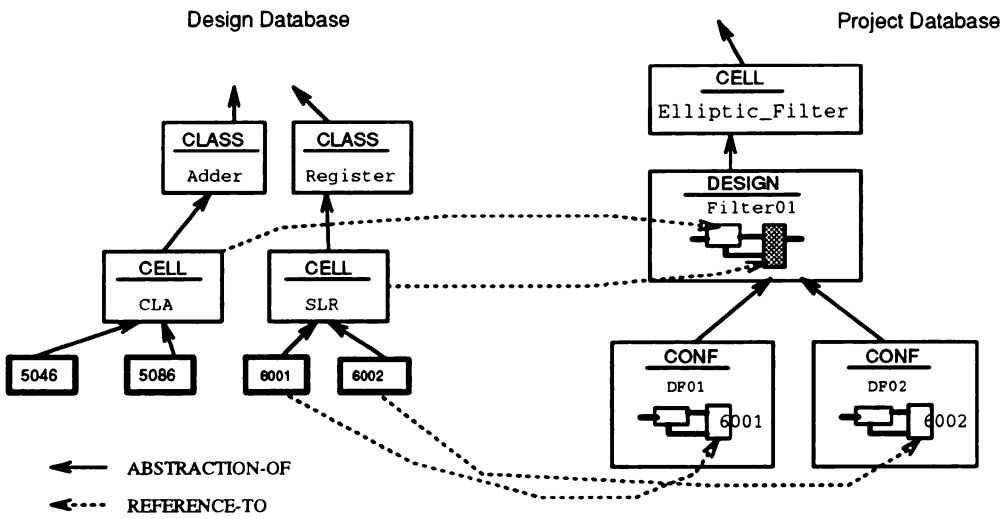


Figure 2.2. An Example of Configuration Binding

2.2.2 Four Levels of Abstraction in Different Domains

A design entity can be specified using an interface and an implementation, as discussed in [9], where an implementation means the contents of a design entity. Since an interface is seen as an *abstraction* of an implementation, it may have different implementations. Furthermore, the design data of an implementation can be separated

into a *generic implementation* and a *configuration.* A generic implementation can be seen as a template for configurations. By a template, we mean a design composed of components such that their interfaces are determined but their implementations are not. In our data model, an interface, a generic implementation, and a configuration are represented by a CELL object, a DESIGN object and a CONF object, respectively.

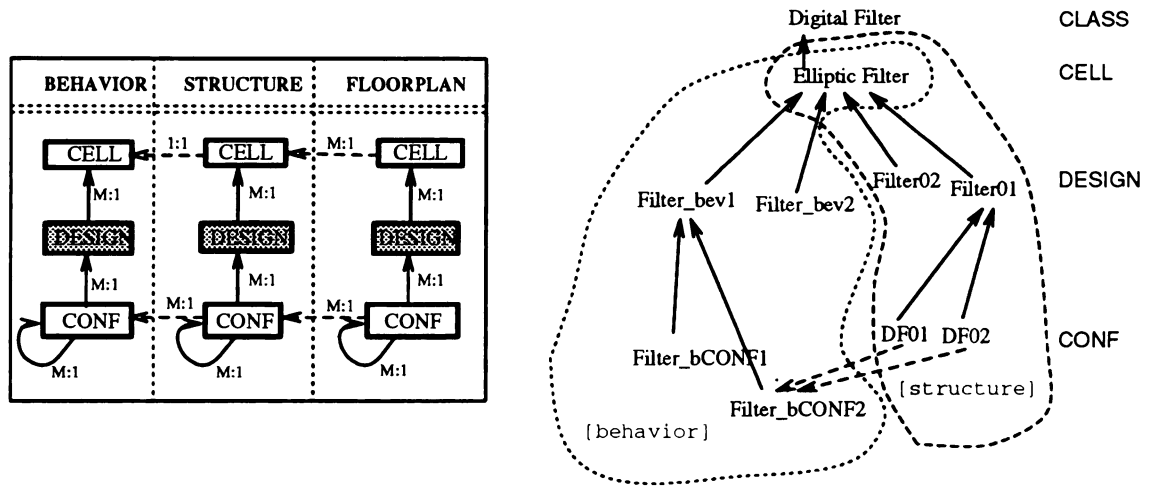
Our concept of "four levels of abstraction" can be applied to any domain if the design entities in the domain are represented based on the concept of separating interfaces and implementations. Figure 2.3 summarizes the data of objects over three different domains. The corresponding VHDL constructs are in parentheses. A VHDL architecture, in either the behavioral or structural domain, is built around the interfaces of smaller design objects (*e.g.*, components). Each of its VHDL configurations contains the bindings of the interfaces to specific implementations. In the layout domain, each CELL object or each DESIGN object does not directly correspond to any VHDL construct. A CELL object in the domain contains the aspect ratio of it and the physical positions of its ports. However, the data of the object can be captured by a VHDL entity and the VHDL attributes attached to its ports. Similarly, the data of a DESIGN object, such as positions and aspect ratios of components, can be represented by a VHDL architecture and the attributes for its components.

Figure 2.4 (a) illustrates the levels of abstraction across the three domains, where solid arrows represent levels of abstraction and dashed arrows represent transformations across domains. Note that each dashed arrow is directed from a less abstract design object to a more abstract design object. Consider a CONF object `A` in the behavioral domain, and a DESIGN object `B` in the structural domain. There is an arrow from `A` to `B`, say "`A <- B`", since design objects in the behavioral domain are more abstract than those in the structural domain. This direction of dashed arrows

	Behavior	Structure	Layout
CELL	ports (entity)	ports (entity)	Shape physical information about ports
DESIGN	behavioral description (architecture)	schematic (architecture)	a collection of components positions of components routes between components
CONF	bindings for components (configuration)	bindings for components (configuration)	bindings for components routes between components

Figure 2.3. Levels of Abstraction at Various Domains

agrees with that of solid arrows. All the arrows of both types are labeled `M:1` for *many-to-one* correspondence or `1:1` for *one-to-one* correspondence. There is a one-to-one correspondence for the CELL-level between the behavioral domain and the structural domain because only one design object is allowed in VHDL for an interface specification over the two domains. Usually, there are many different transformations from an interface specification in the structural domain to one in the layout domain. The selection of the physical attributes of a design, such as shape and pin location, results in a different CELL object in the layout domain. Figure 2.4 (a) illustrates that many-to-one correspondence exists between CELL objects in structural and layout domains. Note that in our data model, the entire data of a design entity is separated into four levels, and CONF objects represent specific design entities. For this reason, the arrow between CONF levels in two domains denotes that a design entity in one domain is transformed into one in the other domain. Generally, once the implementations of the subobjects of DESIGN objects in a domain are determined, giving CONF objects, these CONF objects are transformed into ones in the next domain.



(a) Levels of Abstraction

(b) A Cell Library across Domains

Figure 2.4. Abstraction Hierarchy across Domains

Figure 2.4 (b) illustrates a library of design objects in the behavioral and structural domains. Since CELL objects have a one-to-one correspondence across the domains, there is only one named object, such as `Elliptic Filter`, which designates two CELL objects in the domains. By following solid arrows (*i.e.*, levels of abstraction) and dashed arrows (*i.e.*, transformations), a designer can navigate through a design library built on top of our data model. For example, a CONF object in the structural domain, `DF02`, can be browsed as either one of the structural implementations of `Elliptic Filter` or one of the structural transformations from `Filter_bCONF2`.

Our mechanisms for specifying constraints and using them in configuration binding, described in Chapter 3, are based on the "four levels of abstraction" and developed for design objects in the structural domain. However, they can be generalized to the behavioral and layout domains since that concept still holds true in these two

domains, as described above.

2.3 Data Schema

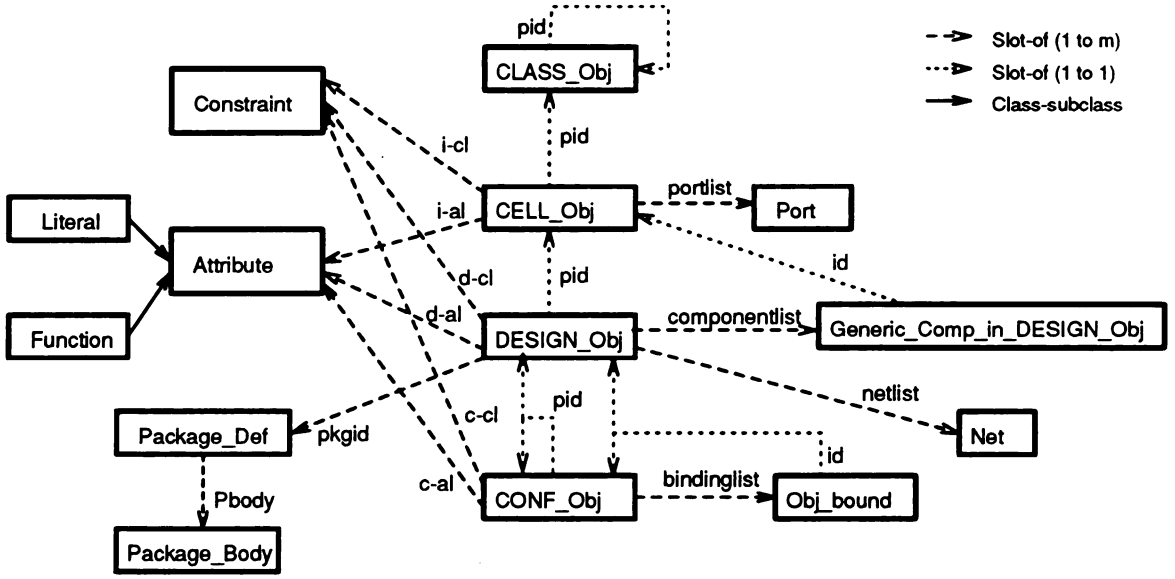


Figure 2.5. Data Schema

Any VLSI object in our data model is an instance of a class defined for its data type. Figure 2.5 illustrates the predefined classes for the data types available in our data model. Like most of the object-oriented data models [3, 8, 39, 78],[¶] we use a fixed number of predefined classes for each data type. In the figure, boxes represent *classes*, solid arrows represent *class-subclass* relationships. Dashed arrows and dotted arrows

[¶]In [39], for example, VLSI objects of the design-entity type, are instances of several classes, such as **logic**, **register**, and **mux**, which are specializations of one root class (called **Cell**).

represent *slot-of* relationships, where the former has 1-to-m correspondences and the latter has 1-to-1 correspondences. Labels beside the dashed and dotted arrows denote slot names. CLASS-level, CELL-level, DESIGN-level, and CONF-level objects are instances of the four classes, `CLASS_Obj`, `CELL_Obj`, `DESIGN_Obj`, and `CONF_Obj`, respectively. Each of the four classes has a slot `pid`, whose value is a parent object. This slot captures the abstraction relationships between design objects. Note that the arrow for the `pid` slot of class `CONF_Obj` is two-headed. This implies that, since incompletely bound CONF-level objects derive other CONF objects, parents of CONF-level objects are either DESIGN objects or CONF-level objects. The slot `portlist` of class `CELL_Obj` contains a collection of port descriptions, which are instances of class `Port`. The slot `componentlist` of class `DESIGN_Obj` contains all of the components to be elaborated. The slot `netlist` of class `DESIGN_Obj` contains a collection of nets. Slot `bindinglist` of class `CONF_Object` contains a collection of component bindings. Each component binding is an instance of class `Obj_bound`, whose slot `id` specifies the design object bound to a component. The value of the slot is an instance of either `DESIGN_Obj` or `CONF_Obj`, implying that a component can remain generic if it is bound to a DESIGN object with its children CONF objects.

CELL-level, DESIGN-level, and CONF-level objects have their attributes in slots `i-al`, `d-al`, and `c-al`, respectively. Similarily, they have their constraints in slots `i-cl`, `d-cl`, and `c-cl`, respectively. Those slots for attributes contain a collection of instances of class `Attribute`. There are two subclasses, `Literal` and `Function`, of the `Attribute` class. The two subclasses represent literal-valued and function-valued attributes, respectively. Since the main objective of our data model is data management for a VHDL design environment, we have classes `Package_Def` and `Package_Body` to represent VHDL package objects and their body objects, respectively[‖] When a DE-

[‖]A VHDL package object is a set of user-defined symbols, such as function names and procedure names. A VHDL package body is a set of definitions of these symbols, such as the body of a function

SIGN object written in VHDL makes use of elements, such as VHDL functions or procedures, in some packages, the `pkgid` slot of the object is a collection of the packages.

For a design object, in our data model, its design data is inherited by its descendants in the following sense. CONF objects represent implementations of design entities, as mentioned earlier. CONF objects keep only component bindings in their slot `bindinglist`. The generic implementation of a CONF object is obtained from slots `componentlist` and `netlist` of its parent DESIGN object. The interface of a DESIGN object is obtained from slot `portlist` of its parent CELL object. The functionality of a design object at CELL, DESIGN, or CONF level is obtained from its ancestor CLASS object. This inheritance between design objects differs from the "inheritance between classes and subclasses", a concept in the object-oriented paradigm. Note that design objects in our data model are *instances* of certain classes. In the object oriented literature [64, 78], two concepts of data modeling have been proposed which support the data sharing between instances. They are "*delegation*"** and "*instance-inheritance relationship*"††respectively. Our data model employs the delegation concept to realize the data sharing between design objects.

---

or a procedure. There may be various VHDL package bodies for one VHDL package object.

**In [64], Parsave, K. *et al.* describe "delegation" as follows. When a message (*e.g.*, a request for accessing slot `portlist`) comes to an object (*e.g.*, a DESIGN object) and the object does not have a method for the message, the message is delegated to another object (*e.g.*, a CELL object).

††In [78], if an instance I_1 has the instance-inheritance relationship with I_2, this denotes that I_2's slots and their value are also defined on I_1.

2.4 Significance of Our Data Model in Building VLSI CAD systems

Classifying data about design entities into four levels provides several advantages for building up VLSI CAD systems:

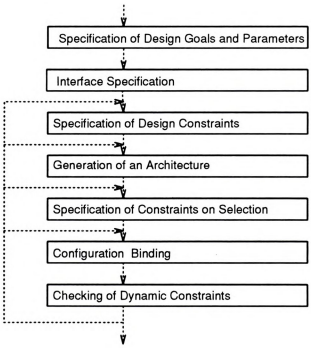


Figure 2.6. A Constraint-Driven Design Methodology

- This classification supports a constraint-driven design methodology, as illustrated in Figure 2.6. In the whole design process, backtracking to previous design points occurs when designers find a partially designed object incorrect. The process proceeds by creating a design object for each abstract level. In

other words, it starts by creating a CELL object, and then creates DESIGN objects and CONF objects. Design constraints to be considered after the interface specification will characterize design goals and design strategies (the selection of algorithms which measure the attributes, such as size and delay, of a design object). Constraints on selection denote the requirements defining the correctness of a configuration under consideration. For example, they include the selection of design objects to be used in configuration binding, upper bounds on specific attributes of selected objects, etc.

- In our abstraction hierarchy, it is easy to locate all the design objects with a given interface or implementation. Since any design written in VHDL is registered as a design object at a certain level, the hierarchy facilitates browsing the VHDL design library. Also, the hierarchy provides a framework for a technology-independent design methodology. In the methodology, CELL-level and DESIGN-level objects are technology independent. We derive a technology-specific CONF object from a DESIGN object by binding its generic components to technology-specific design objects.

- The four levels of abstraction serve as a mechanism for *design-object typing*. A design object can be regarded as a type such that all its descendants are the members of the type. Our typing concept generalizes the concept of *version typing* [9], where design objects with only the same interface form a type.

- Update propagation can be handled easily. The update of an interface should propagate down to its implementations. In our model, interface information is kept in CELL objects. DESIGN objects inherit all the constraints, including an interface, from their parent CELL object. They are always sensitive to updates of the constraints of the CELL objects since the constraints are always enforced on the design of DESIGN objects. Another kind of update occurs to a

design object that is used as a component of other objects. If the structure of the design object is updated, then several of its attributes may have different values. In this case, the update is captured by the composite objects where the updated object is used as one of their components. The reason is that the composite objects may have the constraints that are defined on the attributes of the updated design object as a component.

- Users can have a clear understanding of how they should use CAD applications. Consider a VHDL design environment, SDE, which is equipped with one kind of window for each level of abstraction [21]. Allowable operations of a window are determined based on the abstraction level of an object under design. The window for a DESIGN object provides users with a schematic editing operation. Updating the interface definition of a component is prohibited in such a window, since this operation must be performed in the window for a CELL-level object.

2.5 Conclusion

We presented an object-oriented paradigm for modeling VLSI design objects. Our data model, in which various types of design entities are uniformly treated as objects, is superior to previous object-oriented data models in terms of capturing the semantics of design process and design data. The data model supports incompletely bound CONF objects, provides a methodology for technology-independent design, and facilitates locating design objects with a certain functionality or a set of constraints. It will be shown to provide a framework for efficient version modeling and configuration binding in the following chapter.

CHAPTER 3

A Constraint-Driven Methodology for Configuration Management in VHDL Design Environments

This chapter presents a *constraint-driven* methodology for version modeling and configuration management in VLSI/CAD design environments. Under our object-oriented data model from the previous chapter, the methodology utilizes design constraints to identify various relationships between versions, provide formal definitions of those relationships, and to ensure the correctness of configurations. A technique is also suggested which increases flexibility in configuration binding by using control information about how to enforce constraints in the process.

3.1 Introduction

It is generally acknowledged that a crucial part of VLSI/CAD design environments is data management [81]. VLSI/CAD applications and designers deal with design objects that evolve in multiple dimensions: refinement, revision, composition, and transformation. This is the reason that VLSI/CAD frameworks have placed focus on version modeling [10, 19, 47, 48] and system support for configuration management [7, 22, 47, 51, 60], among their capabilities for data management. Version modeling, which is defining various relationships between versions of design objects, is essential for the understanding of the basic requirements of version control services.

This chapter presents a constraint-driven methodology for version modeling and configuration management under our object-oriented VLSI data model in Chapter 2. In a version model proposed herein, design entities are versioned *separately* at each abstraction level, implying that multiple revisions and alternatives of design objects (interfaces, implementations, and configurations) are created and maintained. Our versioning mechanism not only minimizes the overhead of backtracking to previous design points, but also allows for a richer set of version relationships than other models [9, 10, 47, 81]. Constraints serve as a vehicle for formally defining relationships between versions in our version model. This capability provides a formal framework for representing and identifying the version relationships, such as compatibility and alternative.

The primary task of configuration management is to select a specific version for each generic component of a given object such that the resulting configuration meets design requirements. Previous works on configuration management [7, 10, 47, 60], have limitations to both flexibility and integrity. In other words, designers are forced to follow fixed mechanisms, such as dynamic binding schemes, and to perform con-

figuration binding without any systemic support for error detection [15,22,24]. Our methodology for configuration management exploits constraints to overcome those limitations, and maximizes flexibility and user interaction with CAD frameworks.

We classify design constraints into four categories. Since the work presented herein was originally developed for a VHDL design environment SDE, constraints are specified using VHDL with several additional constructs.

The rest of this chapter is organized as follows. Section 3.2 describes a method for classifying and specifying design constraints. Various relationships between versions are discussed in Section 3.3. Section 3.4 describes a mechanism for constraint-driven configuration binding. Section 3.5 concludes.

3.2 Constraint Classification and Specification

In our data model presented in Chapter 2, design objects can have constraints, which are conditions on their attributes. We classify design constraints into two categories: *dynamic* and *static*. Note that constraints can be seen as predicates defined on attributes of design objects. We define constraints to be static if we can algorithmically compute the values of the attributes on which they defined. Static constraints are further divided into two subcategories: *implicit* and *explicit*. *Implicit* constraints are predetermined based on the semantics of data types so that designers need not specify them. For example, a simple wire should connect two ports with the same width (no. of bits). *Explicit* constraints should be specified by designers. Such constraints are usually related to delays, size, power consumption, etc. *Dynamic* constraints are concerned with the dynamic behavior of a design object, and are verified during simulation. The objects corresponding to constraints are not found in VHDL semantics. We consider only static constraints in checking the correctness of designs.

3.2.1 Four Categories of Constraints

This section describes the explicit constraints needed for ensuring the correctness of design objects. We classify those constraints into four categories as follows.

- *Performance*: Restrictions on performance measures of a design object. The measures of interest for practical designs are size, propagation delay, power consumption, and latency, as mentioned in [80].

- *Environment*: Specifications of the environment where a design object can be used as components. They include fanout restrictions, operating temperatures, etc. Such constraints on a design object are different from constraints from the performance category in that they should be met by any large design with the design object as a component. In this sense, the enforcement of them on a large design is similar to *module import*, a concept well defined in Modula-2 [45], since they are passed to the design along the component hierarchy.

- *Relativity*: Constraints for consistent uses of design objects. Constraints from this category reflect designers' experiences or their preferences about using two different design objects as components of a large design. There are two kinds of relativity constraints: *must-type* and *cannot-type*. A constraint of *cannot-type* (*must-type*) asserts that design objects X_1 and X_2 *cannot* (*must*, respectively) be used as components of a large design at the same time.

- *Selection*: Restrictions on selecting design objects as components in configuration binding. They specify either a specific design object or a group of design objects for a component. The latter denotes that one chosen from the group must be used as a component.

We believe that the above four categories cover all types of constraints for the constraint-driven methodology described in Chapter 2.4. Since constraints are expressed in a uniform way regardless of their categories, as described in Section 3.2.2, it is easy to express the new categories of constraints that would be identified later.

```
configuration  rec_conf of z-method is
--+ assert (1) L1'delay < 40ns; -- constraint A
--+ assert (2) environment'operating_temperature <= 90; -- constraint B
--+ assert (1) from(L1,alu_r) -> from(M2,cache_1) -- constraint C
--+ assert (1) from(L1,alu_s) -> not from(L3,cache_2) -- constraint D
--+ assert (1) from(L1,alu_1) and from(M1, cache_1)) -- constraint E
--+ assert (2) M1'cpu = MC68K; -- constraint F
begin
--   L1, M1, and M2 are the components of rec_conf.
     ...
end
```

Figure 3.1. Constraints written in extended VHDL

3.2.2 Mechanisms for Representing Constraints

This section describes how to specify design constraints, especially those that need to be enforced on configuration binding. We specify constraints by using VHDL with additional constructs. Written in extended VHDL, constraints are in the form of VHDL **assert** statements, and included in the VHDL text as *annotations*. Annotations are commonly used to include additional features into VHDL [4]. Assume that `z-method` is a VHDL architecture of a signal receiving circuit and `rec_conf` is a VHDL configuration of the architecture. Figure 3.1 illustrates example constraints for `radiotrans`. In this example, we will use a design library shown in Figure 3.2. All the lines starting with "`--+`" are constraint specifications, and are syntactically

VHDL annotations. Numbers following the keyword **assert** denote the degrees of significance, which will be described in Section 4.6.

In VHDL, it is possible to attach arbitrary user-definable attributes to design objects. The evaluation of constraints requires the attributes of components, where such attributes are actually attached to the *component types* of those components. The *component type* of a component means the design object to which the component gets bound. Our extended VHDL allows for the *attribute inheritance of a component from its component type*, a feature not found in VHDL. In our extension, for a component `X`, the notation `X'Z` represents an attribute `Z` of `X`. If `X` is bound to a design object `Y`, the above notation actually represents attribute `Z` of `Y`. For specifying constraints from the selection or relativity category, we define a boolean function **from**. Function **from**(`X,W`) is "true" if component `X` is bound to either design object `W` or one of `W`'s descendants.

Constraint *A* from the *performance* category restricts the maximum delay of component `L1` to 40ns. Therefore, `L1` can be bound to design objects only if they have delay < 40ns. The reserved word **environment** denotes any design object that uses as a component a design object under consideration. In this example, the constraint *B* specifies that **`z-method`** can be a component of design objects only if they have operating temperature <= 90. Constraints from the *must-type* relativity category are in the form of "**from**(`X,W`) `->` **from**(`X',W'`)", where operator "->" denotes "*implies.*" Similarly, constraints from the *cannot-type* relativity category are in the form of "**from**(`X,W`) `->` **not from**(`X',W'`)". Based on the library shown in Figure 3.2, constraint *C* specifies that `M2` *must* be bound to one of **`cache_1`**'s descendants if `L1` is bound to **`alu_r`**. Constraint *D* specifies that `L3` *cannot* be bound to **`cache_2`** if `L1` is bound to **`alu_s`**.

Constraints from the *selection* category are represented using the **from** function or

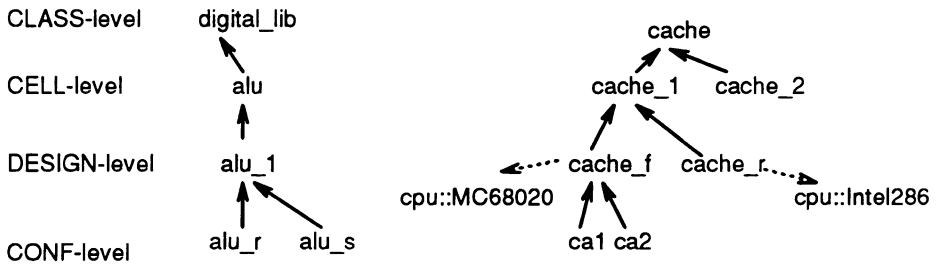


Figure 3.2. An Example of Cell Library

the attributes of individual components. The interpretation of a constraint "**assert from**(X,W)" depends on whether design object W has its descendants or not. Therefore, it depends on the abstraction level of W. The constraint specifies that if W is a design object, for example a CELL object or a DESIGN object, with its descendants, X must be bound to one selected from the descendants of W. Instead, if W has no descendant, component X must be bound to W. In other words, such a constraint from the selection category specifies the *direct coupling* between component X and design object W, as implied by VHDL **use** statements. In our example, a function **from**(L1,alu_1) of constraint F denotes that component L1 must be bound to one of the two CONF objects alu_r and alu_s.

In addition to CLASS objects, we have introduced the notion of a *classification attribute* for further grouping design objects based on various classification criteria. The collection of possible values for a classification attribute forms a hierarchy of specialization-generalizations. Assume that *cpu* is a classification attribute, where its value hierarchy starts with Intelx86 and MC68K. MC68K has two specializations, MC68010 and MC68020, and Intelx86 has also two specializations, Intel286 and Intel386. Constraints on classification attributes of an individual component are

used to further narrow the selection of components, indicated by constraint F of Figure 3.1. The constraint F requires M1 be bound to a design object such that the value of a classification *cpu* of it is either MC68K or a specialization of MC68K. Figure 3.2 shows attribute cpu of two design objects cache_f and cache_r. Constraints E and F in Figure 3.1 specify that component M1 must be bound to either CONF objects ca1 or ca2, since attribute cpu of the objects is MC68020, a specialization of MC68K.

```
--+ assert (1) z-method'size <= 1000;
--+ for size of entity z-method use (size_by_sum(z-method));

--+ function size_by_sum (C: dobj) return integer;
--+ begin
--+      variable I, S: integer; variable temp_id:  dobj
--+      S := 0;
--+      for I := C'Low to C'High
--+      loop
--+           temp_id = C'Comp[I];
--+           S := S + temp_id'size;
--+      end loop;
--+      return S;
--+ end;
```

Figure 3.3. A Function-valued Attribute

Let us consider a constraint defined on a function-valued attribute, as in Figure 3.3. Certain attributes are function-valued ones, whose values are obtained by executing a function. Consider the size of a design object as the total size of its components. For a function-valued attribute, the function for it sometimes needs to be described in a *structure-independent* way. This description style is necessary if a designer wants to attach a function-valued attribute to a design object when the structure of the object is not yet fully determined. For supporting this structure-

independent description, our extended VHDL adds three reserved attributes and one type identifier.

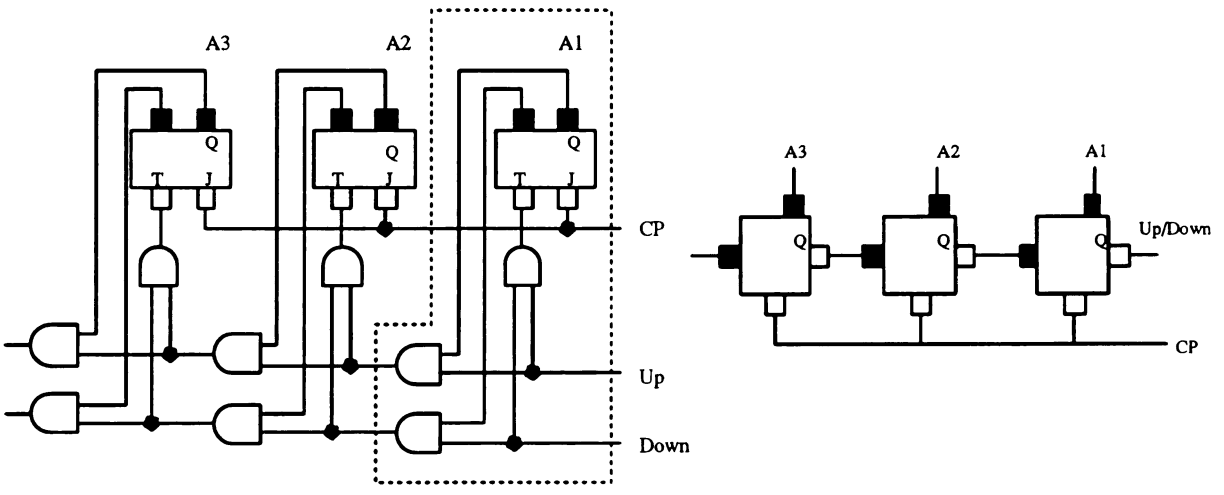
The three attrbibutes of CONF objects and DESIGN objects are: *Comp, Port*, and *Net*. They are arrays of components, ports, and interconnections, respectively. Figure 3.3 illustrates that the value of the *size* attribute is computed by a function *size_by_sum*, as specified by the **for** statement. The result of the function is the total size of the components of `C`, which is given as a parameter. The type identifier, **dobj**, is one for variables whose values are design objects. Two attributes of *Comp* (*Port* and *Net*), *Low* and *High*, are the lowest and highest index of the *Comp* (*Port* and *Net*) array, respectively.

3.2.3 Manipulation of Constraints Using Abstract Objects

For manipulating constraints easily, we have developed the concept of an *abstract object* [21]. An abstract object is an object that realizes an abstraction for any portion of a structural design. Typically, an abstract object is defined as a collection of VLSI objects of a certain data type. Abstract objects help a designer in performing some CAD operations, such as schematic editing and constraint manipulation, as well as specifying constraints by hiding design details. The types of abstract objects are as follows.

Abstract Cell:

An *abstract cell* is an aggregation of design objects, interconnected by wires. Usually, it is composed of a subset of the components of a DESIGN object. Figure 3.4 shows an example 3-bit up/down counter. Let `S` be an abstract cell that denotes the substructure surrounded by dotted lines. Using this abstract cell can simplify schematic editing of the counter in a way that `S` is instantiated three times and then `S`



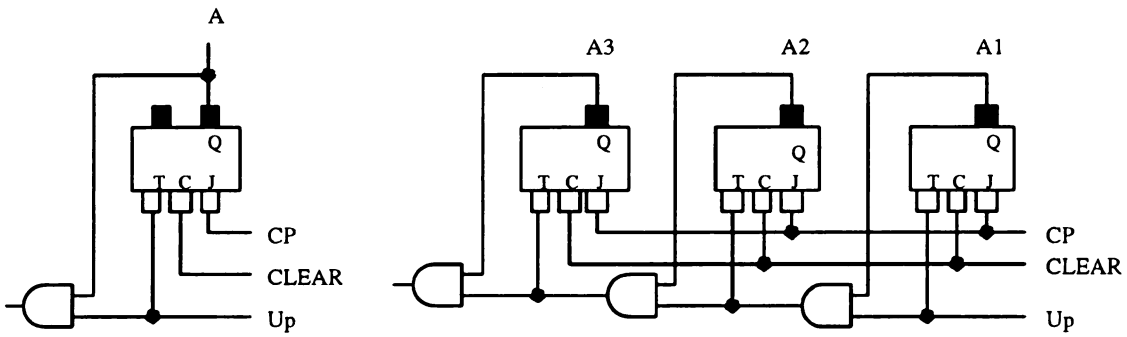
(a) 3-Bit Up/Down Binary Counter (b) Abstract Cell/Abstract Port

Figure 3.4. An Example of Abstract Object

is decomposed into its original structure. Similarly, related nets and related ports can be grouped into *abstract ports* and *abstract nets*, respectively. These abstract objects also simplify wiring components while creating or revising a version of a schematic. For example, the two nets `Up` and `Down`, shown in Figure 3.4, are abstracted into an abstract net called `Up/Down`. Wiring will first be carried out using the abstract wire, and then all the occurrences of the abstract wire can be decomposed into the original two nets.

Using abstract cells can facilitate specifying constraints which are enforced on only a substructure of a schematic `D`. Using abstract cells can simplify the specification of such a constraint. Suppose a constraint that limits the total size of the substructure to 100. To enforce it, first an abstract cell `S1` is defined from the substructure, and an attribute *size* is assigned to `S1`. The value of the attribute can be a function

`space_using_sum`, shown in Figure 3.3, which computes the total size of `D`. Then, the constraint shall be in the form of: "**assert** `S1`'*size* < 100". Specifying the constraint is more difficult if abstract cell `S1` is not used. That is, a specific function should be written which computes the total size of the substructure in such a way that each of `D`'s components is checked to see whether it is within the substructure.



(a) A Slice of 3-Bit Up Binary Counter

(b) 3-Bit Up Binary Counter

Figure 3.5. An Example of Abstract Cell Decomposition

For increased flexibility, we relax the constraint that abstract cells with abstract ports need to be decomposed into their original structure. Note that the main role of abstract ports is hiding detailed descriptions. Thus, an abstract cell with abstract ports can be replaced by any structure, if ths structure can be abstracted into an object that has the same port description as the cell. For example, consider a structure shown in Figure 3.5 (a). The structure can be abstracted into one with the same port description as abstract cells in Figure 3.4 (b). We can obtain the counter shown in (a) of Figure 3.5 after replacing all abstract cells shown in Figure 3.4 (b) with the

structure.

Abstract Constraint:

An *abstract constraint* is a collection of constraints. The concept of an abstract constraint can ease assigning a set of related constraints to design objects. As an example, SDE has two types of interactive windows called *attribute windows* and *constraint windows*. Figure 3.6 illustrates the attributes and constraints of a DESIGN object `ClkGen` (Clock Generator). The design object `ClkGen` is currently associated with only an attribute `space`, whose value is computed using a user-defined function `size_by_sum` written in extended VHDL. This figure shows a pop-up menu composed of menu items for the manipulation of constraints.

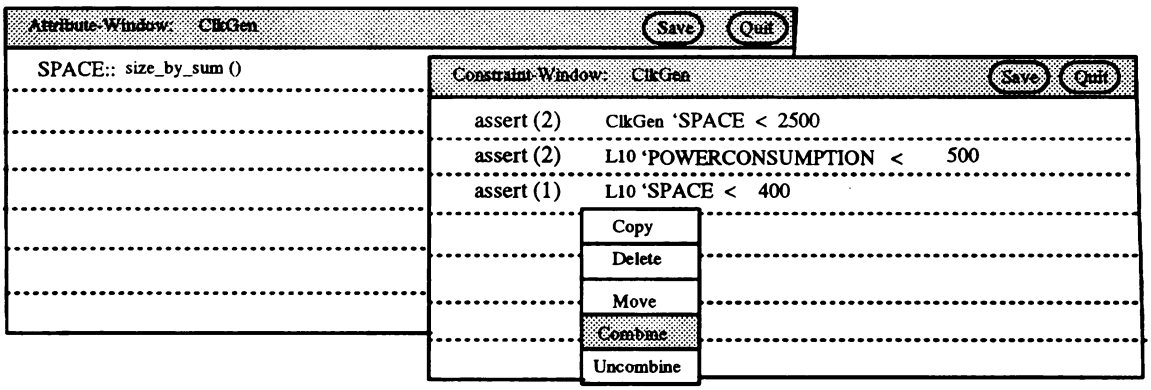


Figure 3.6. Examples of an Attribute Window and a Constraint Window

More than one constraint can be grouped as an abstract constraint. An abstract constraint is treated as a single object for manipulation operations, such as "copy" or "move," so designers need not deal with its constituent constraints individually.

An abstract constraint *power_space* can be constructed from the three constraints of Figure 3.6. Then, **assert** *power_space* will replace the three constraints in the constraint window. Abstract constraints can take parameters. If component `L10` needs to be a parameter, an abstract constraint constructed from the three constraints will be: "**assert** *power_space*(*comp_id*)." The parameter *comp_id* should be replaced by some component when this abstract constraint is assigned to design objects. Two menu items, `Combine` and `Uncombine`, are provided for creating an abstract constraint and decomposing it into its constituents, respectively.

3.3 Version Relationships

A version model presented in this section captures various relationships between the design objects that have evolved in several dimensions. Constraints can serve as a mechanism for grouping related versions. They can also be used to define formally the relationships between design objects. This formalism allows CAD environments to identify some relationships between a pair of versions.

3.3.1 The Version Space of Four Dimensions

In our version model, a design entity evolves in four orthogonal dimensions: *abstraction, revision, changing of constraints*, and *transformation.* A design object at some level in the abstraction hierarchy is seen as a type from which the design objects at lower levels are derived. Modifying a DESIGN object or a CONF object results in a new object, which is a *revision* of the original object. Previous VLSI data models [9, 10, 46, 81] do not distinguish design evolution in abstraction and revision dimensions. The models use one term *versions* to denote the new objects derived in

these two dimensions. Another dimension for design evolution is changing of design constraints, not found in previous VLSI data models [9, 10, 46, 81]. The reason is that a design object should satisfy constraints on its ancestors in our abstraction hierarchy. For a design object, if a designer changes only the *constraints* on the object, a new object is created which differs from the object only in constraints. Then, the new object will have a different set of its descendants from its original object. A design object in one domain is transformed into other domains, as indicated in Figure 2.4 (a).

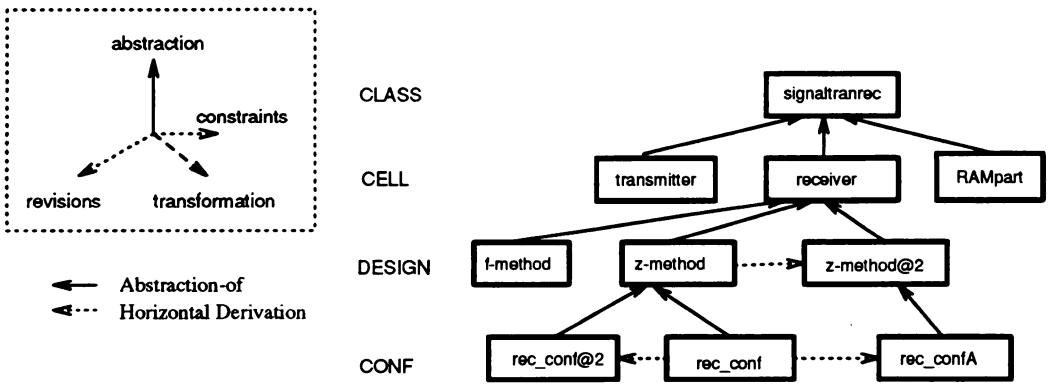


Figure 3.7. Version Evolution

Since a CLASS object is a named group of functionally-related design entities, it does not evolve through general design activities, such as modification, transformation, etc. A group of versions for a design entity, evolving in the four dimensions, conceptually form a space called *version space.* Every design object except CLASS-level objects is located in a version space by a four-element tuple: <*space-name, domain, abstraction-level, object-id*>, where *space-name* designates a design entity,

domain is a domain, *abstraction-level* is a level and *object-id* identifies the design object. Our version model uses version numbering as a mechanism for tracing the evolution history of the design object at the same domain and level. This numbering convention is similar to those of previous version control systems [73]. Therefore, the last element *object-id* is a string designating an initial design object, optionally followed by a version number. For example, for a DESIGN object in Figure 3.7 which evolved from an initial DESIGN object `z-method`, it has the following four-tuple representation: <receiver, structure, DESIGN, z-method@2>.

In the design library based on our data model, the inheritance of constraints along the abstraction hierarchy is a feature that clusters design objects made under the same constraints. This clustering makes it easy to locate all the designs with given constraints. However, this sometimes leads to what we call a *constraint-driven derivation.* For example, a CONF object `rec_confA` is a derivation of another CONF object `rec_conf`, as illustrated in the figure. Suppose that *rec_confA* does not satisfy all the constraints on its parent, `z-method`, so this CONF object cannot be registered as one of its children. Instead, a new derivation `z-method@2` is created, and becomes the parent of `rec_confA`. In this case, we call `z-method@2` a *constraint-driven derivation* introduced by `rec_confA`. The constraints on `z-method@2` will be the same as those on its original object `z-method` except that all the constraints contradicting `rec_confA` are excluded.

3.3.2 Relationships between Design Objects

Let us define notation first. For a design object O, let ANCESTORS(O) be the set of all O's ancestors. By definition, if two design objects O_1 and O_2 have the same parent, ANCESTORS(O_1) is equal to ANCESTORS(O_2). Let LEVEL(O) be the abstraction

level (*i.e.*, CELL-level, DESIGN-level, and CONF-level) of a design object O, and CONST(O) be the set of all constraints enforced on O. CONST(O_1) $\subset$ CONST(O_2) means that the constraints to be satisfied by design object O_2 are stricter than those by design object O_1. We assert CONST(O_1) $\subset$ CONST(O_2) when for each constraint `C` in CONST(O_1), `C` is in CONST(O_2) or `C` subsumes some element in CONST(O_2). When a constraint specification is seen as a logical assertion, the term "subsume" means "generalize" (in other words, "is less restrictive than"), exactly as in predicate logic [17]. For example, consider three constraints C_1, C_2 and C_3: C_1 is "`L10'space < 500`"; C_2 is "`L10'space < 400`"; C_3 is "`L20'space < 300 and L10'space < 400`". Then C_1 generalizes C_2, and C_1 (or C_2) also generalizes C_3.

Our version data model supports the following types of relationships between design objects in the abstraction hierarchy.

- *Alternative*: For a design object, if a group of design objects are its different realizations at some lower abstraction level, they are called *alternatives* with respect to it. Design objects O_1 and O_2 are alternatives to design object O_3 if O_3 in ANCESTORS(O_1), O_3 in ANCESTORS(O_2) and LEVEL(O_1) = LEVEL(O_2).

- *Horizontal Derivation*: Design object O_2 is a horizontal derivation of design object O_1 if O_2 results from modifying O_1 or from changing the constraints on O_1. By definition, ANCESTOR(O_1) is equal to ANCESTOR(O_2).

- *Dependency*: Design object O_1 is dependent on design object O_2 if O_1 *must* be used together with O_2 in large designs.

- *Nonpairability*: Design objects O_1 and O_2 are nonpairable if they *cannot* be used together in large designs.

- *Compatibility*: Design object O_1 is compatible with design object O_2 if O_1 can replace O_2 in large designs. O_1 is *partially* compatible with O_2 if O_1 and O_2

are alternatives with respect to some CELL object and for each constraint $\mathtt{C} \in$ CONST(O_2), C is not contradictory to any element in CONST(O_1). O_1 is *fully* compatible with O_2 if O_1 and O_2 are alternatives with respect to some CELL object and "CONST(O_2) $\subset$ CONST(O_1)" holds.

- *Interchangeability*: Design objects O_1 and O_2 are interchangeable if O_1 is fully compatible with O_2 and O_2 is fully compatible with O_1.

- *Component*: Design object O_1 has a component relationship with O_2 if O_1 is a component of O_2.

- *Hidden version*: Design object O_1 is one of the hidden versions of design object O_2 if there is a configuration binding that derives O_1 from O_2. Hidden versions do not really exist in the design library. Instead, they are implicitly derived from an incompletely bound CONF object or a DESIGN object when needed.

3.3.3 The Detailed Description of Version Relationships

Alternative relationship identifies a group of design objects that can be bound to a component. Suppose that a component of a DESIGN object is an object O. The component is bound to one of design objects that are alternatives with respect to O. We can find easily alternative relationships among design objects by using a graph form of the design library, illustrated in Figure 2.4. Then, design objects O_1 and O_2 in a domain are *alternatives* with respect to design object O_3 if LEVEL(O_1) = LEVEL(O_2) and O_3 is a common ancestor of O_1 and O_2.

The definition of horizontal derivation implies that the versions derived by revision or changing of constraints remain at the same abstraction level as their original object. Horizontal derivations are different from alternatives in that they have the

same parent. In Figure 3.7, for example, three design objects, `rec_conf`, `rec_conf@2`, and `rec_confA`, are alternatives with respect to CELL object `receiver`. However, the horizontal derivation relationship exists between only `rec_conf` and `rec_conf@2`. Since alternative relationships are obtained from abstraction-of and transformation relationships in the library, as mentioned earlier, our version model uses version numbers only for keeping track of horizontal derivations.

Nonpairability and dependency relationships act as relativity constraints in configuration binding. Those relationships between design objects need to be checked for any large design object that has components bound to the design objects. Our definition of the compatibility relationship matches the intuitive notion of *upward compatibility*, a concept widely used in software engineering [65] and hardware configuration management. The separation of the compatibility into the two subcategories, such as partially compatible and fully compatible, is based on the degree of replaceability. It is a popular method of exploring the design alternative space to substitute other design objects for components of an existing design. Compatibility plays an essential role in the substitution because the use of library design objects compatible with the components will preserve design consistency. Since compatibility and interchangeability are formally defined using constraints, versions with these relationships can be automatically obtained.

The concept of hidden versions as well as the four levels of abstraction are features that distinguish our version model from others [10, 69, 71]. Hidden versions are useful for the efficiency of storage. For example, in Figure 3.8, note that CONF object `DF01` (or `DF02`) is incompletely bound if the component `CLA` has not yet been bound. `DF01` could derive two CONF objects, `DF01-1` and `DF01-2`, if the unbound component is bound to `5046` and `5086`, respectively. These two CONF objects are hidden versions if they are assumed to exist in the library. Thus, a designer need not generate a

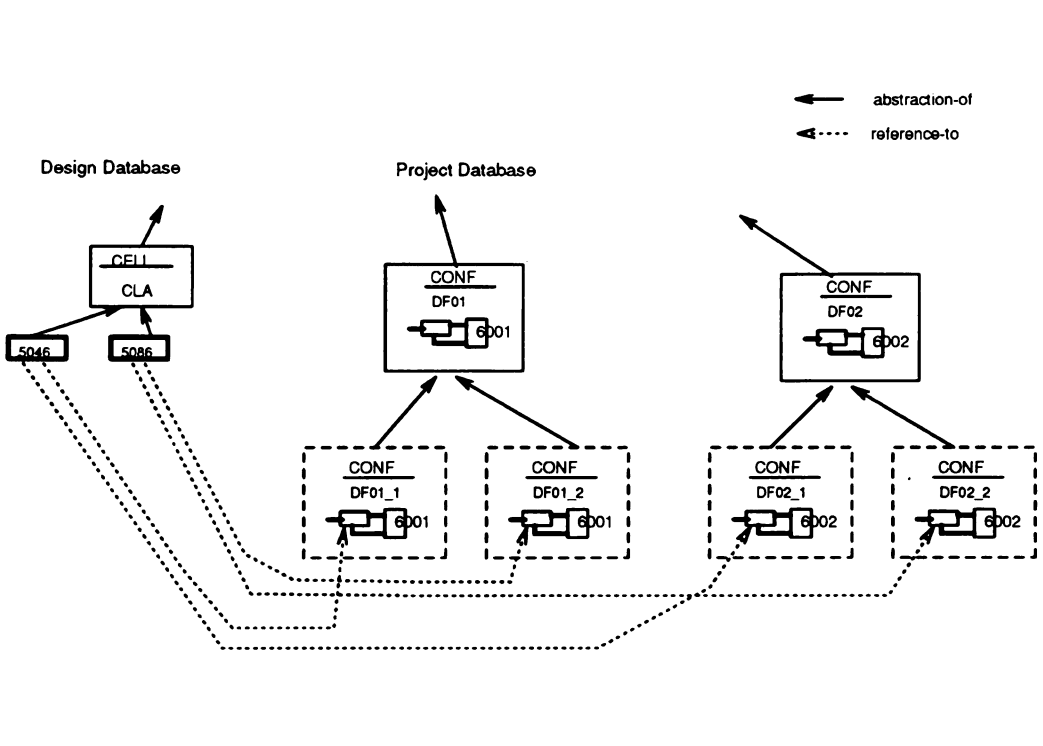


Figure 3.8. An Example of Hidden Version Derivation

lot of CONF objects explicitly. Hidden versions are also exploited in configuration binding to provide a richer set of design objects for generic components. Once a hidden version is selected and bound to a component during configuration binding, a corresponding design object with all its components bound is registered into the library.

The number of hidden versions of a design object grows exponentially with the number of its generic components. Moreover, hidden version derivation can be carried out hierarchically. In other words, deriving hidden versions of a design object may require hidden versions of its components. In order to prevent the proliferation of hidden versions, we employ two simple strategies: hidden versions derivation is allowed only for incompletely bound CONF objects that are leaf nodes in the design library; hierarchical derivation is not performed. The second strategy is partly because once

hidden versions are derived, a designer tends to assure their correctness before they are recursively used in deriving other hidden versions. Also, since hidden versions of a design object must satisfy constraints on the design object, the number of hidden versions can decrease if designers enforce more or stricter constraints.

TYPE	ABSTRACTION LEVEL	INHERITABILITY
alternative	same	No
horizontal derivation	same	No
dependency	same/different	No
nonpairability	same/different	Yes
compatibility	same	No
interchangeability	same	No
hidden version	same/different	No

Figure 3.9. Properties of Relationships

More detailed properties of the relationships are summarized in Figure 3.9. The *domain* property determines whether two design objects involved in a relationship are in different domains or in the same domain. The *abstraction level* property determines whether two design objects involved in a relationship must be at the same abstraction level or can be at different levels. When design object O_1 has a relationship with design object O_2, the *inheritability* property determines whether O_1 still has the relationship with any of the descendants of O_2.

3.4 Configuration Binding

Configuration management is a mechanism for *cell selection*, the process of selecting design objects for generic components of a large design object from the library such that the selected objects can satisfy the constraints on the design object. In our methodology for configuration management, this process is carried out during configuration binding for DESIGN objects or incompletely bound CONF objects. Our methodology also supports dynamic binding, a mechanism not found in VHDL, for dynamically determining a specific design object to be bound to a component.

3.4.1 Dynamic Cell Selection

Generic components can be bound in two ways: *static* and *dynamic*. In static binding, a component is a specific version of a design object. Our methodology for configuration management allows two cases in which static binding occurs: *CONF objects* are derived from a DESIGN object or an incompletely bound CONF object; hidden versions are retrieved and explicitly registered into the library. Our methodology supports dynamic binding as follows. When a design object with generic components is retrieved (or accessed), a specific object is dynamically chosen from the set of candidate design objects for each generic component, and then bound to the component. We call such a specific object a *dynamically chosen version*. In the previous work on cell selection [7, 10, 47, 60], dynamic binding schemes are fixed in that they choose the newest version or a design object with the most recent version index. The following features distinguish our dynamic binding from previous work:

- We determine dynamically chosen versions among all design objects that satisfy various design parameters represented in terms of constraints.

- For increased user interaction, a designer can express arbitrary dynamic binding schemes. Furthermore, a scheme can be applied for an individual component, implying that dynamically chosen versions for different components can be determined using different schemes. This capability gives a designer ample flexibility for configuration binding, not found in [10, 47, 60].

A user-defined dynamic binding scheme is in the form of a VHDL function. A function used as a dynamic binding scheme takes an array of candidate design objects as one parameter, and finds an object with certain properties (for example, one with the smallest delay) from the array, and then returns it. Such an array is associated with each component which is to be dynamically bound, and design environments fill up the array with candidate objects before executing the function. For clarity, the following is the definition of an example function `nv_strategy` with an array *objarray* as its parameter:

```
--+ functionnv_strategy(objarray:  array (integer range <>) of dobj)
 --+     return dobj ;
```

3.4.2 Constraint-Driven Cell Selection

There are several options for making up lists of candidate design objects for generic components. For example, hidden versions can be included into the lists or excluded, depending on these options. To store this kind of control information, a designer can associate a control frame with a design object under configuration binding. Our cell selection is carried out by retrieving the control information from the control frame. The structure of the control frame is shown in Figure 3.10.

Slots	Description
Hidden_Versions	Decision on the inclusion of hidden versions.
Checking_Level	Designating a set of constraints to be temporarily turned off.
Temp_Constraints	Designating a set of constraints to be temporarily enforced.
Incomp_bound_Cells	Decision on the inclusion of incompletely bound objects.
Dynamic_Selection	Designating dynamic binding schems for components.

Figure 3.10. Control Frame

- The slot *Hidden_Versions* specifies whether hidden versions must be included or not.

- A designer can turn off some constraints temporarily during use. For example, a constraint on the maximum size of a design object can be violated during design because the size may exceed the upper bound temporarily during the optimization stage. The slot *Checking_Level* contains a number such that constraints are are ignored during configuration binding if their degree of significance is larger than the number.

- The slot *Temp_Constraints* contains a set of constraints that a designer wants to enforce temporarily during design.

- The slot *Incomp_Bound_Cells* specifies whether incompletely bound objects can also be selected for generic components. By incompletely bound objects, we mean DESIGN objects or incompletely bound CONF objects. In some case, a designer may defer the final binding of a generic component by binding the

component to an incompletely bound object. Later, the designer will bind the component to a more elaborated design object.

- The slot *Dynamic_Selection* specifies dynamic binding schemes for generic components.

3.4.3 A Cell Selection Procedure

A cell selection procedure is as follows. Let O be a design object under consideration in configuration binding. Assume that O has generic components, X_1, X_2, ..., X_n. For each component X_k, the cell selection proceeds as follows. First, we compute S_k, the set of candidate design objects satisfying local constraints on X_k, where the *local constraints* on a component are the constraints whose evaluation is affected by the binding of the component alone. A procedure *ModSel*, shown in Figure 3.11, is called with X_k and O's control frame, and returns S_k. In the control frame, the *Hidden_Versions*, *Incomp_Bound_Cells*, and *Dynamic_Selection* slots specify cell selection options for *individual* components. For convenience, the default option can be specified using the keyword **default**. If these slots contain a pair <**default**, *value*> without a pair <X_k,*value*>, the second element, *value*, of the first pair denotes the option for X_k.

Next, every combination of n design objects, one from each S_k, is examined to determine whether it can satisfy all constraints on design object O. There are an exponentially large number of combinations to be examined. This examining is the dominant factor of the time complexity of the cell selection procedure.

In our methodology, designers have two options. One is that if S_k contains more than one design object, a designer selects one design object from S_k. The other is that all those combinations are enumerated until one combination is found that satisfies all

constraints on the design object O. Moreover, constraints from the selection category can significantly decrease the search space by considering only appropriate design objects.

```
/* Assume that component X_k is a design object ctype */

ModSel (X_k, control_frame)

Step 1) Compute the set C of local constraints c on X_k such that the
        checking level of c is less than or equal to Checking_Level of
        control_frame.

Step 2) If a constaint from(X_k, W) is in C, then set r to W, else set r
        to ctype.
        If r is a completely bound CONF object, then
             if r satisfies C, then return a set consisting of only r, else
             return ∅.
        else go to step 3.

Step 3) Compute the set D of descendants of r (including r) in the library.
        If slot Incomp_bound_Cells of control_frame specifies yes for X_k, then
             compute S from design objects of D which satisfy C.
        else
             compute S from design objects of D which are completely bound
             objects and satisfy C.

Step 4) If slot Hidden_Versions of control_frame specifies yes for X_k, then
             for all incompletely bound CONF objects of S, compute set S' of
             all their hidden versions which satisfy C, and then
             add S' to S.

Step 5) If slot Dynamic_Selection of control_frame specifies a function f for X_k,
             then call function f with S, and return an object selected by the
             function.
        else return S.
```

Figure 3.11. Procedure *ModSel*

3.4.4 An Example of a Cell Selection

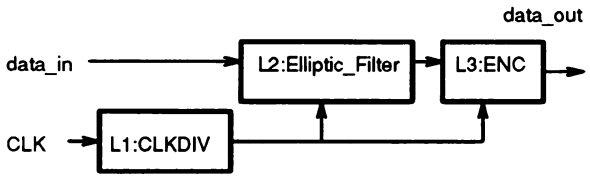
For clarity, an example of cell selection is illustrated in Figure 3.12. DESIGN object DSP1 under configuration binding is a VHDL architecture of a VHDL entity DSP. DSP1 consists of three generic components: L1 (a clock divider), L2 (an input filter), and L3 (an encoder). These three components are design objects CLKDIV, Elliptic_Filter, and ENC, respectively. Following the VHDL syntax, "S => V" in Figure 3.12 denotes value V of slot S. Let us assume that DSP1 has no constraints inherited from its ancestors. Let C_1, C_2, and C_3 be constraints such that C_1 is ''(1) L3'*POWERCONSUMPTION* <= 50"; C_2 is ''(1) **from**(L3,05)"; C_3 is ''(1) DSP1'*SPACE* <= 125". Then C_1, C_2, and C_3 are are all the constraints on DSP1, since C_3 is in the *Temp_Constraints* slot and the checking levels of the three constraints are less than the *Checking_Level* slot. In the figure, 01, 03, 04, and 05 are completely bound CONF objects. As illustrated in Figure 3.8, CONF objects DF01-1 and DF01-2 are the hidden versions of DF01, and DF02-1 and DF02-2 are the hidden versions of DF02.

Consider L1 first. There is no local constraint for this component. Only the CONF objects, 01, 03, and 04, need to be considered, since the slot *Incomp_Bound_Cells* specifies that incompletely bound design objects, such as 02 and CLKD1, must be excluded. Note that (L1, nv_strategy) is in the slot *Dynamic_Selection.* Procedure nv_strategy, chooses the newest version. Then, S_1 becomes {01}, since 01 is the newest of 01, 03, and 04. Consider L2 next. The component has only one local constraint C_1. Since the slot *Hidden_Versions* specifies "yes" for all components, the hidden versions of the two incompletely bound CONF objects, DF01 and DF02, must be considered. Those hidden versions are DF01-1, DF01-2, DF02-1, and DF02-2. Then, S_2 is {DF01-1, DF02-1}, since they are completely bound design objects and can satisfy C_1. Lastly, consider L3. This component needs to satisfy one local constraint

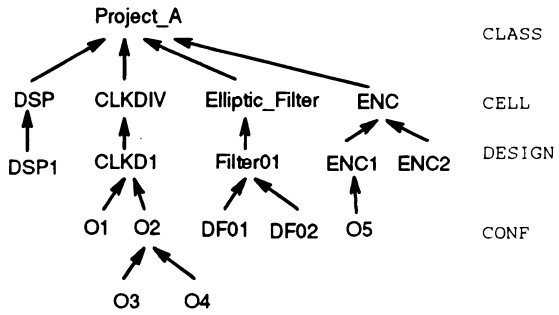
C_2. The constraint denotes that CONF object `O5` is a sole candidate design object for the component. Therefore, S_3 is {`O5`}. The next step is to find the solutions satisfying constraint C_3. There is only one solution, which is {`O1, DF01-1, O5`}.

3.5 Conclusion

We presented a version modeling and a constraint-driven methodology for configuration binding. Using features of the object-oriented paradigm, such as inheritance, our version model organizes design objects in four dimensions, and can capture a richer set of relationships between versions, including hidden versions, than the previous work. Configuration binding is carried out in a constraint-driven fashion so that the constraints enforced by designers are checked in order to validate design consistencies at early stages. The approach also offers a method for specifying options of cell selection, and increasing designers' control over the enforcement of constraints.



[Logic Diagram of DSP1]



[Example Cell Library]

```
architecture DSP1 of DSP is
   --+ (1) L2'POWERCONSUMPTION <= 50;
   --+ (1) from(L3, O5)
begin
 --+ for control_frame of entity DSP1 use
 --+  (Hidden_Versions => (<default, yes>),
 --+   Checking_Level => 2,
 --+   Temp_Constraints => ("(1) DSP1'SPACE <= 125"),
 --+   Incomp_Bound_Cells => (<default, no>),
 --+   Dynamic_Selection => (<L1,nv_strategy>))
 ...
end
```

[Control Frame]

cell	power cons.	size	update time
O1	40	55	Aug/8
O2	-	-	
O3	45	60	Aug/2
O4	60	75	June/22
O5	40	30	
DF01_1	45	45	
DF01_2	60	57	
DF02_1	50	56	
DF02_2	60	60	

[Attribute Values]

Figure 3.12. An Example of Module Selection

CHAPTER 4

A Path-Oriented Algorithm for the Cell Selection Problem

An algorithm for the *cell selection problem* is presented. Given a network G of logic gates, the problem is to select a library cell for each gate such that the longest delay through G is at most T_{max} and the total area of selected cells is minimum. We first prove the *strong* NP-completeness of the problem if the circuit is a general acyclic graph. The proof implies that there exists no pseudo-polynomial time algorithm unless P=NP. We next propose a path-oriented, heuristic algorithm that iteratively chooses *paths of gates* with delay larger than T_{max}, and then selects cells for the paths. The set of paths chosen induces a maximal series-parallel subgraph. We also present a *cloning* method which further improves a solution obtained by the path-oriented algorithm. In the proposed cloning method, nodes are duplicated into clones such that the resulting graph becomes a series-parallel graph. Our cloning method is more efficient in terms of time and space than the tree cloning of Chan [16].

The results of our algorithm are compared to those of Lin's work [57] and misII [31]. The algorithm provides significantly better solutions to all the circuits than the

previous work.

4.1 Introduction

Given a network G of logic gates and a cell library, the cell selection problem* is to select a list of cells (called a *binding*) for G which minimizes the total area of the selected cells and restricts the longest delay to the maximum allowable delay T_{max} [23]. The cell library contains multiple cells with discrete sizes for each gate type. If the library is a collection of standard cells, the cell selection problem corresponds to a cell-based transistor sizing problem [25, 29, 70]. Technology mapping is generally defined as a covering problem of all the matches of cells† over the network. If there are multiple cells of the same Boolean function with different performances, the cell selection can follow technology mapping to minimize the total area of all the matches by selecting a cell for each match.

There has been research on the cell selection problem [16, 57, 58]. Lin [57] and Li *et. al.* [58] proposed heuristic methods which iteratively improve a binding of the network by using cell replacement. Chan [16] presented an algorithm that transforms the graph into an equivalent tree by duplicating gates of the graph. Since the number of newly added gates is exponentially large, the algorithm works only for relatively small-size circuits. Pseudo polynomial time algorithms have been proposed to find an optimal solution when the network is restricted to a tree or series parallel graph [16, 58] and when the fanout delay is not considered. The fanout delay of a cell in a network is the delay due to the load capacitance of the cell. However, it is not known whether there exists a pseudo-polynomial time algorithm for the problem when the network is a general graph.

In this chapter, we prove that the time complexity of the problem for a general

*The cell selection problem was also named "the optimal selection of standard cells" [57] or "the circuit implementation problem" [58].

†Each match is a subgraph of the network. Cells are represented by graphs.

graph is *strongly NP-complete*, implying that there is no pseudo-polynomial time algorithm for the problem unless P=NP. For general networks with fanout delay, we propose a path-oriented algorithm which iteratively chooses a set of paths, and then selects cells for the gates on these paths until the longest delay is less than equal to T_{max}. The algorithm employs a method of selecting paths whose new binding will result in the largest decrease in the longest delay without significantly increasing the total area.

Our path selection method is based on finding a maximal series-parallel subgraph that includes all those paths. In the previous heuristic approaches [57, 58], a small number of gates are considered to improve the current binding for each iteration. Therefore, the heuristics are more likely to reach a local minimum than our algorithm which improves the area globally by considering as many gates as possible. Since all of the gates on a critical path may not be chosen at an iteration, a new binding of the path can be far from an area minimum selection.

A *cloning* method is also presented which further improves an initial solution obtained by the path-oriented algorithm. In this method, we construct a series-parallel graph (called a *cloned series-parallel graph*) from a given network G by duplicating some nodes of G into clones, and obtain a binding of the cloned graph.

The structure of this chapter is as follows. In Section 4.2, the cell selection problem is formulated and a dynamic programming algorithm for a series-parallel graph is presented. In Section 4.3, we prove the strong NP-completeness of the cell selection problem. Our cell selection algorithm for a general graph appears in Section 4.4 and Section 4.5. We tested our algorithm with the LGSynth91 logic synthesis benchmark circuits [82], and compared the results with Lin's work [57]. Section 4.6 describes experimental results. We conclude in Section 4.7.

4.2 Preliminaries

In this section, we first formulate the cell selection problem, and then present a cell selection algorithm for a series-parallel graph. An acyclic network can be modeled as a Directed Acyclic Graph (DAG) in such a way that nodes and edges represent gates and interconnections, respectively. We will use two terms *network* and *graph* interchangeably. Also, we use terms *node* (*edge*) and *gate* (respectively, *interconnection*) interchangeably.

4.2.1 Formulation of the Cell Selection Problem

Let G be a DAG of m logic gates, g_1, g_2, g_3, ..., g_m, and let T_{max} be the maximum allowable delay of the network. For each g_i of the DAG, L_{g_i} is the set of library cells that can realize (*i.e.*, be bound to) gate g_i. Library cells differ in area, internal delay, input capacitances, and driving capabilities. Each cell c in L_{g_i}, $1 \leq i \leq m$, has discrete area denoted by $A(c)$.

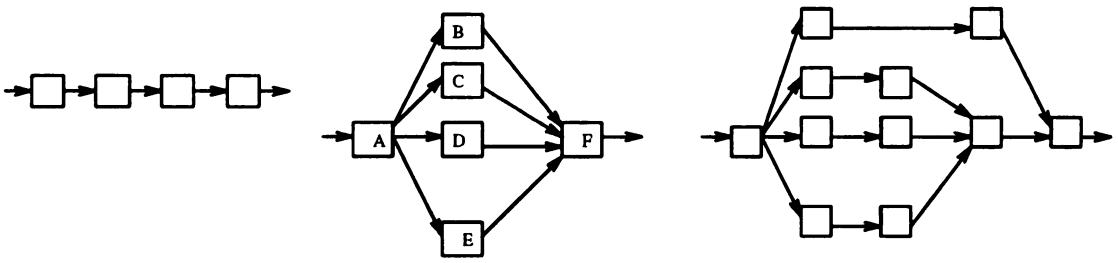
The *cell selection problem* for G is to select a list of cells (*i.e.*, a binding of G), c_1, c_2, ..., c_m, and bind each c_i to g_i, $1 \leq i \leq m$, such that

1. $c_i \in L_{g_i}$,
2. S is minimized subject to $T \leq T_{max}$, where S is the total area of selected cells, and T is the longest delay through G.

For simplicity, each cell has multiple input ports and one output port. In the network, there is only one driver that drives a signal to an input port of a gate. Our delay model is basically identical to the delay calculation method specified in the LGSynth91 logic synthesis benchmark [82]. The term $D(c, g)$ is the delay of cell c

when cell c is bound to a gate g. $D(c, g)$ is sum of the *internal delay* and the *fanout delay* of cell c. The internal delay of cell c, denoted by $d(c)$, is not affected by the capacitive loading of c. We assume that a cell has one internal delay between all its input ports and its output port. The output port of a cell has an attribute *load factor*, which is the capability of driving an output signal. The unit of load factor is delay per unit load capacitance. Each input port of a cell has a capacitance, called an input capacitance. The *load capacitance* of a cell is the total input capacitance of the ports to which the cell drives its output signal. When cell c is bound to gate g, the fanout delay of c is calculated by the product of the load factor of c and the load capacitance of c.

Consider a path of cells $< c_1, c_2, ..., c_m >$ which are bound to gates $g_1, g_2, ..., g_m$ respectively. The *delay of the path* is computed by $\sum_{1=i}^{m} D(c_i, g_i)$. For node u, let us call a node driving an input signal of u a *fanin node* of u. Let us call a node driven by the output signal of u a *fanout node* of u.



(a) a chain (b) a graph containing a set of parallel nodes (c) a series-parallel graph

Figure 4.1. Examples of series-parallel graphs

Let us define several graph-theoretical terms. A graph G is a *two-terminal graph* if there are two nodes, a *source* node and a *sink* node, such that the source node has no fanin node and the sink node has no fanout node. A series-parallel graph is recursively defined as follows [75] (a series-parallel graph is a two-terminal graph): (i) A DAG consisting of two nodes joined by a single edge is a series-parallel graph. (ii) If G_1 and G_2 are series-parallel graphs, so are the series-parallel graphs constructed by each of the following operations (1) and (2): (1) *Parallel Composition*: Identify the source node of G_1 with the source node of G_2 and the sink node of G_1 with the sink node of G_2. (2) *Series Composition*: Identify the sink node of G_1 with the source node of G_2.

In a series-parallel graph $G(V, E)$, we define set $S \subset V$ to be a set of *parallel nodes* with respect to two nodes v_1 and v_2 if each $u \in S$, v_1 is the only fanin node of u and v_2 is the only fanout node of u.

Figure 4.1 illustrates examples of series-parallel graph. A *chain*, constructed by series compositions only, is shown in (a). Parallel compositions are shown in (b), in which nodes B, C, D, and E are a set of parallel nodes with respect to node A and node F. A general series-parallel graph is shown in (c).

4.2.2 A Cell Selection Algorithm for A Series-Parallel Graph Under Fanout Delay Effect

This section describes a dynamic programming algorithm that finds a near optimal binding of a series-parallel graph when the fanout delay is considered. In [58], Li *et al.* proposed an algorithm that finds the optimal binding of a series-parallel graph when cells are assumed to have *no* fanout delay. Their algorithm is motivated by the observation that a series-parallel graph is a composition of two types of subgraphs,

chains and sets of parallel nodes. We will explain how their algorithm can be extended to the case where we consider fanout delay. Note that if the output port of g_i is connected to an input port of g_j, then the delay of g_i depends on the binding of g_j. Thus, we must compute the binding of the network in the *backward* direction rather than *forward*.

Consider a chain of length m, g_m, g_{m-1}, ..., g_1, such that g_m is the *first node*, g_{m-1} the second node, and so on. Let F_k^i denote the area of the optimal binding of node g_1 through node g_i such that the delay of the binding is at most k. Then, F_k^i is given by the following formula:

$$F_k^i = \min_{c \in L_{g_i},\ k' + D(g_i,c) \leq k} \{F_{k'}{}^{i-1} + A(c)\}$$

For m parallel nodes $g_1, g_2, ..., g_m$ and their fanout node g_{m+1}, the binding can be computed as follows. We first compute $F_k^i[c']$ for each $c' \in L_{m+1}$, where $F_k^i[c']$ is the area of the optimal binding of the m parallel nodes such that the longest delay of the binding is at most k when cell c' is bound to node g_{m+1}. The reason for fixing g_{m+1} at a certain cell is that the input capacitance of g_{m+1} is necessary for the delay calculation of g_i, $1 \leq i \leq m$. $F_k^i[c']$ can be obtained by:

$$F_k^i[c'] = \min_{c \in L_{g_i},\ D(g_i,c) \leq k} \{F_k^{i-1}[c'] + A(c)\}$$

We next compute F_k, the area of the optimal binding of the graph induced by $m+1$ gates, $g_1, g_2, ..., g_m, g_{m+1}$, such that the longest delay of the binding is at most k. The following formula shows the recurrence equation of F_k:

$$F_k = \min_{c' \in L_{g_{m+1}},\ k' + d(c') \leq k} \{F_{k'}^m[c'] + A(c')\}$$

A binding of a series-parallel graph is obtained by repeatedly finding a subgraph, either *a chain* or *a set of parallel nodes and their fanout node*, and then replacing it with its corresponding block. This reduction is repeated until there is only one node left. The corresponding block of a subgraph can be regarded as a node in the reduced graph. If the subgraph is a chain of m nodes, then the library cells for its corresponding block have area F_t^m and delay t, for $t \leq T_{max}$. Similarly, the block of parallel nodes with their fanout node has library cells each with area F_t and delay t.

Figure 4.2 illustrates how the series-parallel graph in (a) is reduced to a single node. In Figure 4.2 (b), nodes J and $A5$ form a chain while J is the fanout node of a set of parallel nodes, $A2$, $A3$, and $A4$. In this case, we can combine J with either of the two subgraphs for the reduction. In this example, node J is used as the fanout node of the parallel nodes.

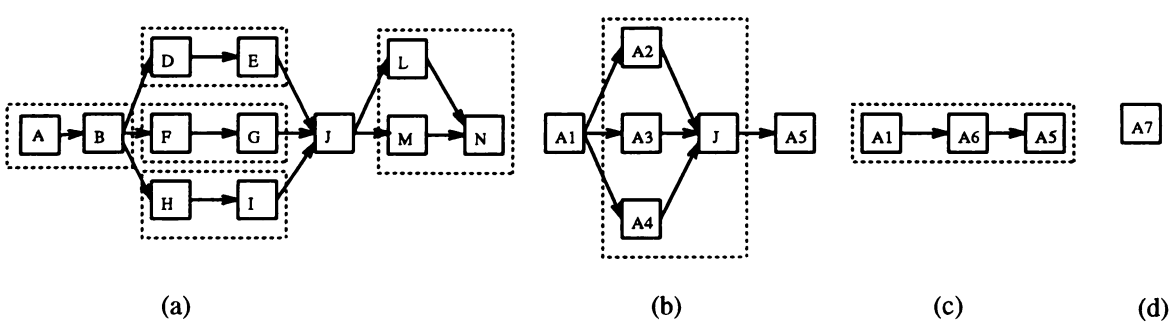


Figure 4.2. A sequence of replacements for a series-parallel graph

4.3 Strong NP-completeness of the Cell Selection Problem

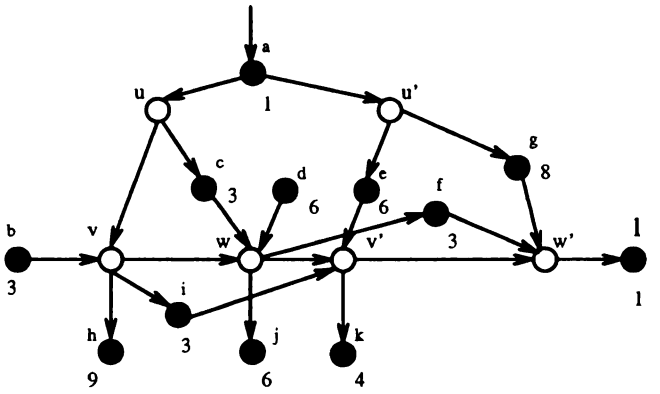
Li *et al.* [58] proved that the cell selection problem is NP-complete by showing a reduction from the partition problem. However, the partition problem can be solved by a pseudo-polynomial time algorithm[‡]based on a dynamic programming approach [36]. Indeed, the algorithm presented in [58] for the cell selection problem in a chain uses a similar dynamic programming approach, and runs in a pseudo-polynomial time. It is not known whether such a pseudo-polynomial algorithm can be extended to a general network.

We prove that the cell selection problem is *strongly NP-complete* for a general graph even when each cell has delay 1 or 3. Thus, finding a pseudo-polynomial time algorithm for the cell selection algorithm in a general network is not possible unless P=NP.

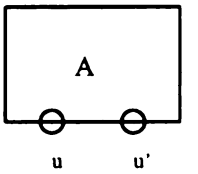
In the proof we use a reduction from the 3-Satisfiability (3SAT) problem. Let $W = C_1C_2...C_r$ be a well formed formula (wff) in a conjunctive normal form over variables $x_1, x_2, ..., x_n$. From W, we shall construct a network G_W with maximum allowable delay T_{max} and maximum allowable area A_{max} such that G_W has a binding with the longest delay at most T_{max} and total area at most A_{max} if and only if W is satisfiable.

In G_W, there are two types of gates, type O, and type T. A gate type O has one library cell with delay 1 and area 1. A gate type T has two library cells a_1 and a_2 with delay(a_1) = 1, delay(a_2)=3, area(a_1)=2, and area(a_2)=1. A delay element with

[‡]An algorithm for a problem A is *pseudo-polynomial* if it solves any instance I of A in time bounded by a polynomial in the size of instance I and the largest integer appearing in I [36]. Thus, if there is a pseudo-polynomial algorithm for the cell selection problem, then the algorithm will find a solution in polynomial time if the weight and area of library cells are not exponentially large.



(a) widget A used for each variable



(b) compact representation of the A widget

Figure 4.3. Variable Structure

delay w can be implemented by a chain of gates of the type O with length w. In the following figures, a black node with weight w denotes a delay element with delay w. and all the white nodes are gates with type T. Note that all black nodes have only one choice of binding, and the total area occupied by them is constant. Thus, we consider the area of only the white notes in each binding.

We construct a directed graph G_W that is composed of two types of widgets, which are pieces of of graphs that enforce certain properties. The first widget, used for each variable, is the subgraph A shown in Figure 4.3. Consider a binding of A with the longest delay 15 and area 9. If we bind a_2 to u, then we must bind a_1 to both v and w; if we bind a_2 to u_i', then we must bind a_1 to v_i' and w_i'. Note that any binding of the structure A must have area at least 9 in order to achieve longest delay at most 15. I and II in Table 4.1 are the only two bindings with the longest delay 15 and with area 9. In the table, III shows another binding with longest delay 15 but requires area 10. IV, V, VI, and VII show bindings with area 9 but their longest

Binding	Delay of White Nodes						Longest Path	Delay	Area
Examples	U	U'	V	V'	W	W'			
I	3	1	1	3	1	3	$< a, U, V, i, V', W', l >$	15	9
II	1	3	3	1	3	1	$< a, U', e, V', k >$	15	9
III	3	3	1	1	1	1	$< a, U', e, i, V, k >$	15	10
IV	1	1	1	3	3	3	$< d, W, V', W', l >$	16	9
V	1	1	3	3	1	3	$< b, V, i, V', W', l >$	16	9
VI	1	1	3	3	3	1	$< b, V, W, V', k >$	16	9
VII	1	1	3	1	3	3	$< d, W, f, W', l >$	16	9

Table 4.1. A set of binding examples

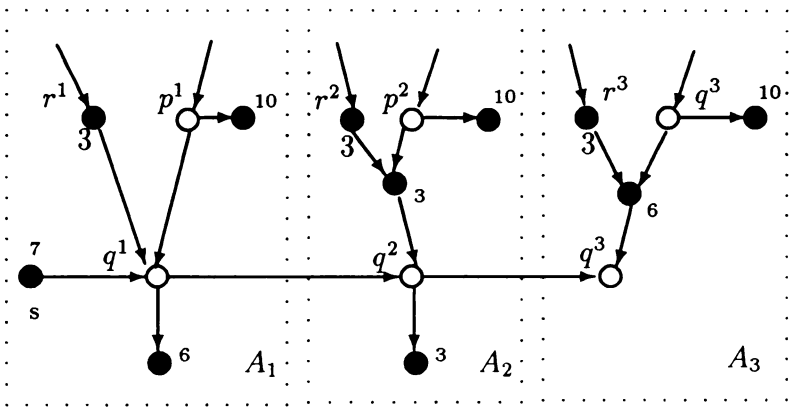
delays are 16. What we shall simulate using widget A for a variable is: the variable is assigned true (false) in any assignment if and only if node u in the widget is bound to a_1 (respectively, a_2).

Another widget B, used for each clause, is shown in Figure 4.4. Each widget has three substructures corresponding to three literals of a clause. Each substructure has nodes r^k, p^k, and q^k, $(k = 1, 2, 3)$. Suppose that the following condition 1 is true for every k $(k = 1, 2, 3)$.

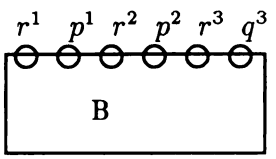
> Condition 1: either (i) it takes delay 4 for a signal to arrive at p^k and delay 2 to arrive at r^k, or (ii) it takes delay 2 for a signal to arrive at p^k and delay 4 to arrive at r^k.

For case (i), a_1 must be bound to p^k to meet the maximum delay 15. For case (ii), the maximum delay for a signal to arrive at q^k (through node r^k) will be 7 for $k = 1$, 10 for $k = 2$ and 13 for $k = 3$. Thus, a_1 must be bound to q_k to meet the maximum delay. Note that if q_1, q_2 and q_3 are all assigned a_2, then the longest delay will be 16 (the delay from s to q_3).

The graph G_W that we shall construct consists of copies of these two widgets. For each variable x_i, we include a copy A_i of widget A, and for each clause C_j, we include a copy B_j of widget B. For any node n in either widget, let n_i denote the copy of



(a) widget B used for each clause



(b) compact representation of the widget B

Figure 4.4. Clause structure

node n in the ith copy of the widget. For example, u_i is the copy of node u in the ith copy (which corresponds to variable x_i) of widget A, and p_j^k is the copy of node p^k in jth copy of widget B. Now we connect the copies of widget A and B as follows. If the k-th literal of C_j is x_i, then connect u_i with p_j^k, and u_i' with r_j^k. If the k-th literal of C_j is x_i' then connect u_i' with p_j^k, and u_i with r_j^k. For example, Figure 4.5 shows the network G_W for $W = C_1C_2$, where $C_1 = (x_1 + x_2' + x_3')$ and $C_2 = (x_1' + x_2' + x_3)$. What we shall simulate using A_k (k=1,2,3) in widget B for a clause is: the k-th literal of the clause is true (false) in any assignment if and only if q^k is bound to a_1 (respectively, a_2).

Now we can claim the followings.

Lemma 1. A wff W is satisfiable if and only if there is a binding of G_W with area $9n + 9 \times r$ and maximum allowable delay 15.

Proof: Suppose W is satisfiable. Let f: $\{x_1, ..., x_n\} \rightarrow \{true, false\}$ be an assignment which makes W satisfiable. For each x_i, if $f(x_i) = true$ then bind a_1 to u_i and a_2 to u_i'; if $f(x_i) = false$ then bind a_2 to u_i and a_1 to u_i'. All remaining nodes

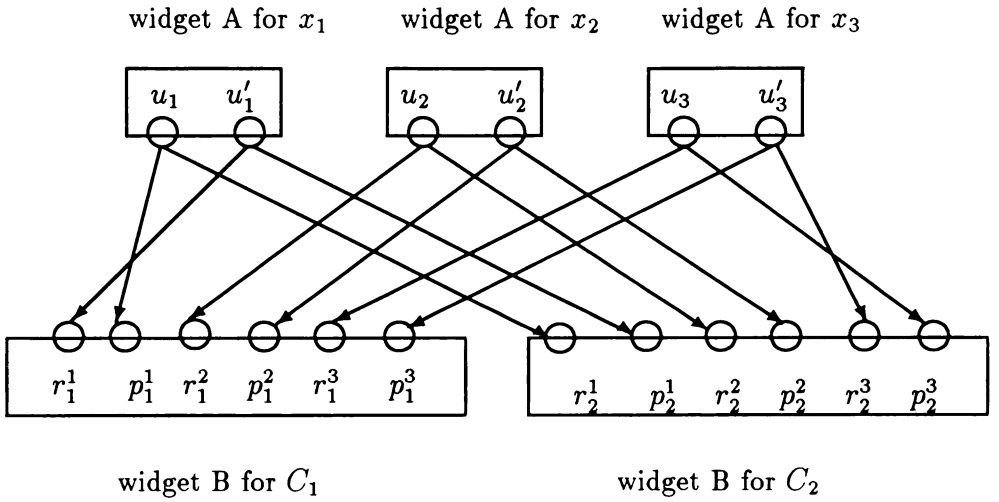


Figure 4.5. An example W = $(x_1 + x_2' + x_3')(x_1' + x_2' + x_3)$

of A_i can be bound accordingly such that the area of A_i is 9 and longest delay is 15. For each clause $C_j = l_1 + l_2 + l_3$, if a literal l_k is *true* under assignment f, then bind a_2 to p_j^k and a_1 to q_j^k; otherwise bind a_1 to p_j^k and a_2 to q_j^k. Under this binding, the total area of G_W is $9n+9r$. Since W is satisfiable, for each clause C_j, there is at least one literal that is true. Thus, the longest path from s_j to q_j^3 is at most 15. Note that under this binding, one of the cases described in the condition (A) is true. Thus the longest delay of G_W is 15.

Conversely, suppose that G_W has a binding with area at most $9n + 9r$ and delay at most 15. Then each A_i must have at least area 9 to meet the maximum delay. Thus, each B_j has area at most 9 to meet A_{max}=$9n + 9r$. That is, for each A_i, either u_i or u_i' must be assigned to a_1. If u_i is connected to p_j^k (that is, x_i is the kth literal of clause j) then u_i and q_j^k must have the same binding to meet the maximum delay. Note that, for each j, at least one of q_j^1, q_j^2, and q_j^1 must be assigned to a_1 to meet the maximum delay 15. Thus, any assignment that assigns x_i to *true* if a_1 is bound to u_i and *false* if a_2 is bound to u_i, makes W satisfiable. □

From Lemma 1, we have the following theorem.

Theorem 1. The cell selection problem is strongly NP-complete. That is, the problem is NP-complete even when each gate has two library cells with delay either 1 or 3.

Proof. The cell selection problem is NP is trivial. The NP-hardness comes from Lemma 1. In the reduction, each gate has only two choices, and each library cell has either delay 1 or 3. □

4.4 A Cell Selection Procedure for Finding an Initial Solution to a General Graph

Our cell selection algorithm for a general DAG consists of two parts. In the first part, we iteratively pick several *paths of gates* and then obtain a new binding of the paths until the longest delay is smaller than or equal to T_{max}. In the second part, we further improve the initial solution by a cloning method. This section describes the first part of the algorithm. The second part will be described in Section 4.5.

4.4.1 Algorithm Description

A procedure for finding an initial solution, named *First_Cell_Selection*, is shown in Figure 4.6.

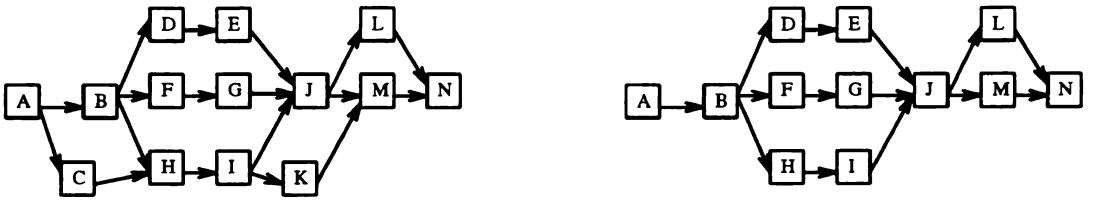
In Step 1.1, each redundant edge $< v, w >$ is removed from G if there is another path from node v to node w. In Step 1.2, we repeatedly identify a *reducible subgraph*, which is either a chain of nodes or a set of parallel nodes and their fanout node, and then replace it with the corresponding block as described in Section 4.2.2.

```
/* G is a network of gates. T_max is the maximum delay. */
Algorithm First_Cell_Selection (G, T_max)
begin
    Step 1(Preprocessing)
        1.1 Remove redundant edges.
        1.2 Obtain an irreducible graph by replacing each reducible subgraph
            with a single node.
    Step 2 (Initial Binding) Bind the smallest (i.e. minimum area) cell to each
        node of G.
    while (the longest delay through G is larger than T_max)
    begin
        Step 3 (SubGraph Construction) Find G', a maximal series-parallel
                                       subgraph of G.
        Step 4 (Cell Selection) Obtain a binding B of G'.
                    Nodes not in G' are bound as in previous iterations.
    end
end First_Cell_Selection
```

Figure 4.6. Procedure *First_Cell_Selection*

In Step 2, we initially bind the smallest cell to each node of G. Steps 3 and 4 are repeated as long as the longest delay under the current binding of G is larger than T_{max}. In Step 3, we choose a set of paths of gates which has the following two properties: (i) the binding of the paths needs to be changed to satisfy the delay constraint T_{max}, and (ii) the paths can induce a *maximal series-parallel subgraph* of G. A maximal series-parallel graph is a series-parallel subgraph such that if any edge is added to the subgraph, the resulting graph is no longer a series-parallel subgraph. The reason for finding a *maximal* series-parallel subgraph of G is obvious: if more nodes and edges are considered for replacement, a new binding of G is likely to be closer to the optimal.

In Step 4, we find a binding $\mathcal{B}$ of G' using the algorithm described in Section 4.2.2, where G' is the series-parallel graph constructed in Step 3. Note that the fanout delay of a node u in G' depends on the load capacitance of u, which is the total



(a) A Network

(b) A Maximal Series-Parallel Subgraph

Figure 4.7. An example network

input capacitance of all u's fanout nodes. In computing the load capacitance of u, we consider all its fanout nodes in G, not restricted to G'. That is, the input capacitance of each fanout node not in G' is added to the load capacitance of u. In this case, the input capacitance will be that of the cell currently bound to the node. For example, consider node A in Figure 4.7. Although the graph of (b) contains only node B, the load capacitance of A is the sum of the input capacitances of nodes B and C. Thus, the input capacitance of the cell currently bound to C is added to the load capacitance of A.

After Step 3 and Step 4 are executed, some paths not included in G' may become critical (*i.e.* delay $> T_{max}$). If this occurs, Step 3 and Step 4 are repeated.

There may be a case when a node selection is oscillating in Step 3: a node was first *selected*, then became *unselected*, and eventually is *reselected*. For such node u, the cell with the largest delay is removed from L_u. Thus, for a node whose selection oscillates as the while loop iterates, the cells for the node are removed from the library in decreasing order of their internal delays. This removal enables the loop to terminate.

4.4.2 A Method for Finding a Maximal Series-Parallel Subgraph of a Network

In the procedure *First_Cell_Selection*, if the longest delay of the network under the current binding is larger than T_{max}, Step 3 is applied to find a maximal series-parallel subgraph. The subgraph must contain as many nodes as possible whose binding affects the longest delay and area.

For a DAG $G(V, E)$, we construct a maximal series-parallel subgraph of G by picking up a set of edges from E (the set of edges in G). Edges are picked up in nonincreasing order of their *weights*. This section first describes a method for computing the weights of edges, and then how to find such a subgraph of G.

Weights of Edges

For a given a binding of the network, the weight of an edge is computed based on the following three factors.

1. Slack: The slack of an edge is defined to be an amount of time by which the longest delay through the edge is larger than T_{max}. When we select edges, edges on a path with delay much larger than T_{max} should have higher priority than those on a path with delay slightly larger or smaller than T_{max}.

2. Heaviness: The heaviness of node u is defined to be the average size of library cells for u. Consider two nodes that differ in heaviness considerably. Since the heavier of the two nodes affects the total area more significantly than other node, the binding of the heavier node is more important than that of the other node. We define the heaviness of an edge to be the average heaviness of the two nodes incident to the edge.

3. Sensitivity: The sensitivity of node u is defined to be the average area-to-delay ratio of cells in L_u. This area-to-delay ratio measures an amount of area increment per unit delay of the binding of u. If we change the binding of nodes with high sensitivity to reduce the delay of a critical path to T_{max}, we can expect on average a smallest area increment. Similar to heaviness, the sensitivity of an edge is defined to be the average sensitivity of the two nodes incident to the edge.

For edge $e = < u, v >$, let $\mathcal{S}(e)$, $\mathcal{H}(e)$, and $\mathcal{Y}(e)$ denote its slack, heaviness, and sensitivity, respectively. For node w, let $h(w)$ and $y(w)$ denote its heaviness and sensitivity, respectively. More formally, the factors are defined as follows.

$$\begin{aligned}
\mathcal{S}(e) &= T(e) - T_{max} \\
\mathcal{H}(e) &= \frac{h(u)+h(v)}{2} = \frac{1}{2}\left(\frac{1}{|L_u|}\sum_{c\in L_u} A(c) + \frac{1}{|L_v|}\sum_{c\in L_v} A(c)\right) \\
\mathcal{Y}(e) &= \frac{y(u)+y(v)}{2} = \frac{1}{2}\left(\frac{1}{|L_u|}\sum_{c\in L_u} \frac{A(c)}{D(u,c)} + \frac{1}{|L_v|}\sum_{c\in L_v} \frac{A(c)}{D(u,c)}\right)
\end{aligned}$$

where $T(e)$ is the delay of the longest path that passes through edge e.

The weight of an edge is obtained from the normalized values of the three factors. To normalize $\mathcal{S}(e)$, $\mathcal{H}(e)$, and $\mathcal{Y}(e)$, we simply divide them by $\mathcal{S}_{max}$, $\mathcal{H}_{max}$, and $\mathcal{Y}_{max}$, respectively, where $\mathcal{S}_{max}$ is the maximum slack of the edges in the network, $\mathcal{H}_{max}$ the maximum heaviness, and $\mathcal{Y}_{max}$ the maximum sensitivity. The weight of edge e, denoted by $W(e)$, is given by the following formula:

$$W(e) = \alpha\frac{\mathcal{S}(e)}{\mathcal{S}_{max}} + \beta\frac{\mathcal{H}(e)}{\mathcal{H}_{max}} - \gamma\frac{\mathcal{Y}(e)}{\mathcal{Y}_{max}}$$

In the above formula, α, β, and γ are constant parameters, which shall be determined experimentally. We used the following values in our experiment: $\alpha = 1$, $\beta = 0.7$, and $\gamma = 0.5$. Note that the larger $\mathcal{S}(e)$ is, the less efficient edge e is. This is the reason that we have a minus ("-") sign in front of $\gamma \frac{\mathcal{Y}(e)}{\mathcal{Y}_{max}}$.

Construction of a Maximal Series-Parallel Subgraph

The following procedure finds a maximal series-parallel subgraph of graph $G(V, E)$.

Subroutine *Find_Max_Series-Parallel_Subgraph* (G)
begin
Step 1: Initialize S to be the collection of edges that are on the longest and the second longest path.
Step 2: For all edges of G not in S, compute their weights.
Step 3: Consider the edges in nonincreasing order of their weights. When an edge is considered, add it to S if the resulting S can still lead to a series-parallel graph.
Step 4: Remove all unnecessary edges from S.
Step 5: Add a dummy source and a dummy sink node to G.
end *Find_Max_Series-Parallel_Subgraph*

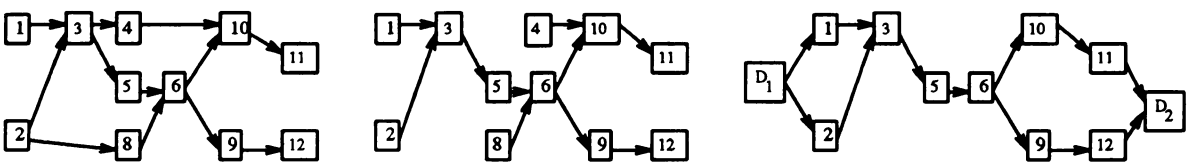
Figure 4.8. A Procedure to Find a Maximal Series-Parallel Subgraph

In the procedure, S contains a set of edges of G. In Step 1, S is initialized to the edges of two paths, the longest and the second longest path through G. Note that the union of any two paths leads to a series-parallel graph in a way that we add one dummy source and one dummy sink node to the union. In Step 2, we compute the weights of the edges of G not in S by using the method described in Section 4.4.2.

To minimize the area increment for decreasing path delays greater than T_{max}, we need to find a set of edges of *maximum total weight* which induces a series-parallel

graph. The problem of finding such a set is NP-complete by reducing the edge-deletion problem to the problem [5]. Therefore, we have developed a heuristic method for finding such a set, as described in Step 3 through Step 5.

In Step 3, edges are considered in nonincreasing order of their weights. For each edge, we should check whether set S can still lead to a series-parallel graph after adding the edge to S. This checking can be done in linear time by using the algorithm of Valdes [75]. In Step 4, unnecessary edges, which are not on paths from primary inputs to primary outputs, are removed from S. In Step 5, S is augmented with a dummy source and a dummy sink node to induce a series-parallel graph. A *dummy node* has a library cell with both delay and area zero.



(a) A Network G (b) S after Step 3 (c) S after Step 5

Figure 4.9. An example of finding a series-parallel subgraph

Figure 4.9 illustrates how to construct a maximal series-parallel subgraph. The graph in (b) illustrates the set S after Step 3. In Step 4, node 4 is eliminated from S since edge $< 4, 10 >$ is not on any path from a primary input to a primary output. Similarly, node 8 is eliminated from S. A series-parallel subgraph as shown in (c) is obtained after adding two dummy nodes D_1 and D_2 to S.

4.4.3 Complexity Issues I

Let us discuss the time complexity of the procedure *First_Cell_Selection*. Consider a network $G(V, E)$. Let L be $\bigcup_{u \in V} L_u$. Step 1.1 is equivalent to finding the transitive closure of G, which takes polynomial time. Step 1.2 can be done in pseudo-polynomial time as described in Section 4.2.2. In Step 2, the smallest cell for each gate can be found in $O(|L|)$ time. Step 3 consists of two parts: computing the weights of edges and sorting them. To compute the edge weights, the longest path which uses e must be found for each edge e. This step can be performed in $O(|V| + |E|)$ time by considering the nodes in topological and reverse topological order [41]. Thus, the weight computation takes $O(|V| + |E| + |L|)$ time. The sorting of weights can be done in $O(|E| \times \log|E|)$ time. The worst time complexity of Step 4 is $T_{max}^2 \times |V|$, which is pseudo polynomial.

For every $|V|$ iterations of while loop, there is at least one node such that the node was first *selected*, then *unselected*, and eventually *reselected*. For such node u, we remove the cell with the largest delay from L_u. Therefore, the while loop requires at most $|L| \times |V|$ iterations in total.

Our algorithm handles only integer delay values, including T_{max}. If a delay value is a real number (as usual), not an integer, the algorithm takes the integer part after multiplying the delay value by an integer M. As a result, M determines the precision of delay values. The larger the number of significant digits we need in representing delay values, the larger M we should use. As shown in Section 4.6, in the general case, the precision loss of delay values minimally affects quality of the solutions.

4.5 A Cloning-Based Improvement Method

In this section, we present a cloning-based method for improving the initial solution of graph $G(V, E)$, resulting from the procedure *First_Cell_Selection*. *Cloning* is a process of duplicating nodes and spreading their in-coming or out-going edges over the duplicated nodes (called *clones*). In our proposed method, graph G is transformed into a *cloned series-parallel graph* by cloning, and then a binding of the cloned graph is obtained using the algorithm described in Section 4.2.2.

Chan [16] proposed a cloning-based cell selection technique, in which a *cloned tree* is constructed from the graph by repeatedly duplicating every node into as many clones as the number of its fanout nodes. Therefore, the number of clones of a node grows exponentially with the depth of the node in the graph. However, in our cloning method, a cloned *series-parallel graph* is constructed and the number of nodes (clones) added increases just linearly with the number of nodes of a graph. When a node is cloned several times, the binding of the node must be consistent among all its clones. Chan's algorithm uses an exhaustive search mechanism which chooses the same cell for the clones for enforcing this consistency. In our method, the binding of the clones is fixed to the solution obtained in Section 4.4. From this improvement method, we shall obtain a new binding of the nodes not being cloned, which improves their old binding.

4.5.1 Basic Concepts and a Cloning Operation

Let us define basic concepts and terms of the cloning operation. The details of our improvement method will be described in Section 4.5.2.

Definition of Terms

Let $G(V, E)$ be a directed acyclic graph (DAG). For nodes w and v of G, w is a *predecessor* of v if there is a path (of length at least 0) from w to v. Especially, if the length of the path is 1, w is called an *immediate predecessor* of v. If w is an *immediate predecessor* of v, then v is a *immediate descendant* of w. For $V' \subset V$, w is a *nearest common predecessor* of V' if 1) w is a predecessor of u for all $u \in V'$ and 2) x is a predecessor of u for all $u \in V'$ and w is a predecessor of x implies that $w = x$. Note that, for a set of nodes, more than one nearest common predecessor may exist.

Definition 1: Given a DAG $G(V, E)$, a node in V is a *join node* if it has more than one in-coming edge. A node in V is a *fork node* if it has more than one out-going edge.

Definition 2: Let $G(V, E)$ be a two-terminal DAG and v be a join node. For node w, which is a nearest common predecessor of all v's immediate predecessors, let V' be the set of all the nodes that are on paths from w to v. The subgraph induced by set V' is called a local graph (*L-graph*) of v (which depends on w). Node $x \in V' - \{v, w\}$ is an outward node (*O-node*) of the L-graph if x has an outgoing edge to a node not in V'.

For example, in Figure 4.10, node 8 is a join node with its immediate predecessors 5 and 7. The nodes 5 and 7 have only one nearest common predecessor, which is node 2. Thus, there is one L-graph $G'(V', E')$ of node 8, where $V' = \{2, 3, 4, 5, 6, 7, 8\}$ and $E' = \{< 2,3 >, < 2,5 >, < 3,4 >, < 4,6 >, < 6,7 >, < 7,8 >, < 5,8 >\}$. The L-graph G' contains two O-nodes, nodes 4 and 6.

The following two lemmas show the properties of the L-graph.

Lemma 2: Let $G(V, E)$ be a two-terminal DAG and $v \in V$ be a join node. If no predecessor of v is a join node, then there is only one L-graph of v.

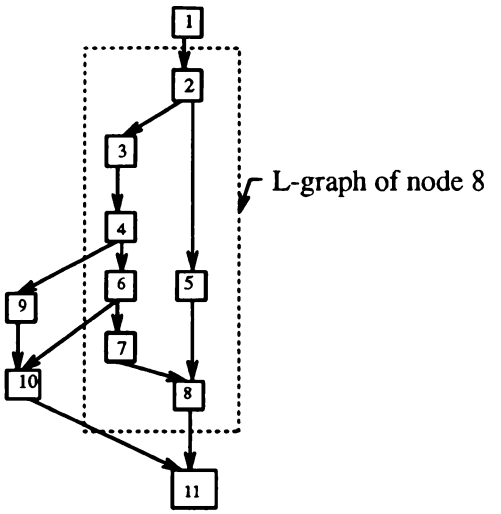


Figure 4.10. An example L-graph

Proof: Since no predecessor of v is a join node, there is only *one* nearest common predecessor for all v's immediate predecessors. Thus, there is only one L-graph. □

Lemma 3: Let $G(V, E)$ be a two-terminal DAG and $v \in V$ a join node such that no predecessor of v is a join node. If there does not exist an O-node is in the L-graph G_v of v, then G_v is a series-parallel graph.

Proof: Note that there is only one nearest common predecessor w of all v's immediate predecessors since no predecessor of v is a join node. Let $G_v(V_v, E_v)$ be the L-graph of v. Since there is no O-node in G_v, for every x in V_v such that $x \neq w$, $(x, y) \in E$ implies that $(x, y) \in E_v$ for any y. We prove the lemma by induction on n, where n is the number of fork nodes of G_v. For n=1, the only fork node in G_v is w, and G_v can be constructed from chains by parallel compositions. Suppose that the lemma holds for all $n < k$. We will prove the case when $n = k$, where $k \geq 2$. Let $u_1, u_2, ..., u_r$ be the immediate descendants of w. For each u_i, define D_i to be the

collection of all nodes of G_v which are descendants of u_i. If x is a common descendant of u_i and u_j, then x is a join node. Since G_v does not have a join node except v, $D_i \cap D_j = \{v\}$ for all $i \neq j$. Since $n \geq 2$, there is some l such that D_l has at least one fork node. Consider the subgraph H induced by $(V_v - D_l) \cup \{v\}$. H has fewer than k fork nodes. By inductive hypothesis, H is a series-parallel graph. Note that the subgraph induced by D_l is a series parallel graph, and the subgraph H' induced by D_l *cup* $\{w\}$ is also a series-parallel graph. G_v is a parallel composition of H and H'. Therefore, G_v is a series-parallel graph. □

Lemma 3 implies that we can apply the cell selection algorithm of Section 4.2.2 to the L-graph $G_v(V_v, E_v)$ of the join node v if V_v has neither an O-node nor a join node except v.

A Cloning Operation

Let $G(V, E)$ be a two-terminal directed acyclic graph. Consider a join node v of G such that no predecessor of v is a join node. By Lemma 2, there is only one L-graph of v. Let $G_v(V_v, E_v)$ be the L-graph of v, and let u be an O-node of G_v. Since no predecessor of v is a join node, u has only one immediate predecessor, say x. Let $\{y_1, y_2, ..., y_k\}$ be the immediate descendants of u which are not in V_v, and $\{z_1, z_2, ..., z_r\}$ be the immediate descendants of u which are in V_v. Cloning u is carried out by duplicating u into two clones, say u_A and u_B, and then spreading the $k + r$ outgoing edges of u over u_A and u_B. More formally, $G(V, E)$ is updated as follows after cloning u:

$$V \leftarrow V \cup \{u_A\} \cup \{u_B\} - \{u\}.$$

$$E \leftarrow E \cup \{< x, u_A >\} \cup \{< u_A, y_1 >, < u_A, y_2 >, ..., < u_A, y_k >\}$$
$$\cup \{< x, u_B >\} \cup \{< u_B, z_1 >, < u_B, z_2 >, ..., < u_B, z_r >\}$$

$$- \{< x, u >\}$$

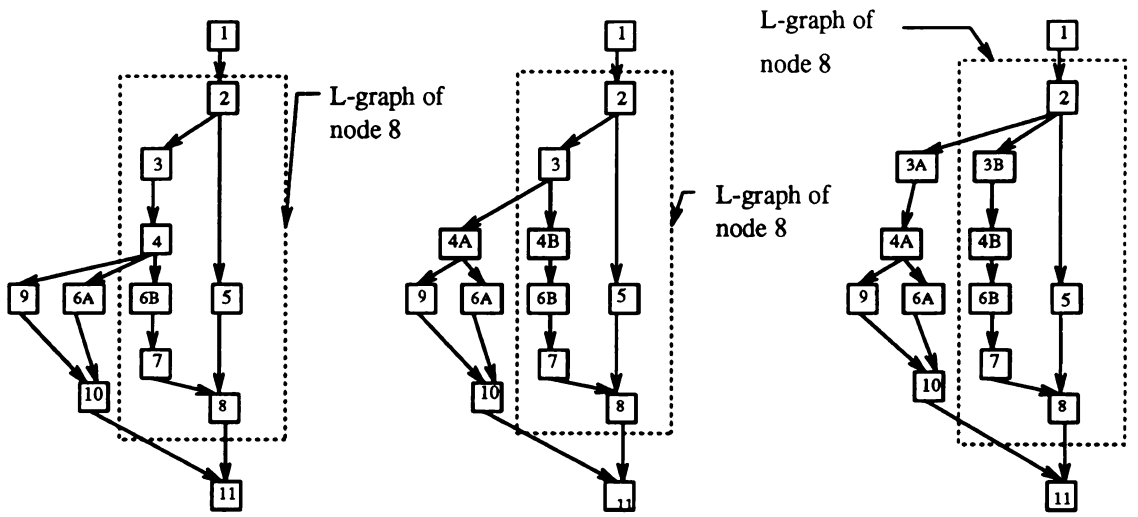
$$- \{< u, y_1 >, < u, y_2 >, ..., < u, y_k >\} - \{< u, z_1 >, < u, z_2 >, ..., < u, z_r >\}$$

The L-graph G_v is also updated accordingly. G_1 will not be in G_v and u_B will be in G_v. Since a node u_A together with an edge $< x, u_A >$ has been added to G as the result of cloning u, x (which is the immediate predecessor of u) will become a new O-node if x is not the source node of G_v (a nearest common predecessor of all v's immediate predecessors). To remove all O-nodes from the L-graph of v, we visit O-nodes of the L-graph repeatedly in reverse topological order and apply cloning operations to them.

Figure 4.11 illustrates how to eliminate O-nodes from the L-graph of node 8 shown in Figure 4.10. The L-graph in (a) has been obtained from the L-graph in Figure 4.10 by duplicating an O-node 6 into clones $6A$ and $6B$. The L-graph in (b) has been obtained from the L-graph in (a) by duplicating an O-node 4 into clones $4A$ and $4B$. Note that cloning a node may introduce a new O-node, which is the parent of the cloned node. In this example, node 3 becomes a new O-node after cloning node 4. The L-graph in (c) is obtained from the L-graph in (b) by duplicating node 3 into clones $3A$ and $3B$.

4.5.2 Cloning and Cell Selection

Our cloning-based improvement method is basically comprised of two tasks: constructing a cloned series-parallel from a given network G and performing cell selection for the cloned graph. If we try to perform all the cloning operations to obtain a series parallel graph, the resulting graph may have an exponentially large number of nodes. To avoid the exponential growth, the two tasks are interleaved as follows. After cloning the L-graph of each join node, the dynamic programming algorithm in



(a) after cloning node 6 (b) after cloning node 4 (c) after cloning node 3

Figure 4.11. An example of cloning

Section 4.2.2 is performed on the cloned L-graph. As a result, the cloned L-graph (which is a series parallel graph) can be reduced to a single node with corresponding library cells.

When a node is cloned into several clones, the binding of the node must be consistent among its clones. To enforce this consistency, if u is cloned to u_1, u_2,..., u_r, then the binding of the r clones is fixed to the binding of u, which is produced by the algorithm in Section 4.4.

Figure 4.13 shows a procedure, named *Improvement_by_Cloning*, which implements the cloning-based improvement method. Step 0 is a preprocessing step whose process is similar to that of Steps 1 through 3. That is, we visit all join nodes of G, apply cloning, and reduce cloned L-graphs to blocks until there is only one node. The only difference is that in the preprocessing step, we do not find bindings of the blocks. Even though some node n of G is not cloned directly, the binding of n must be fixed if a block to which n is reduced is cloned later. Therefore, the binding of n will be used for all clones of n. Step 2 is carried out using the operation described in Section 4.5.1. Step 3 is to apply the algorithm of Section 4.2.2 for the L-graph of join node v. In this step, it should be noted that the area of clones must be counted only once.

Figure 4.12 illustrates an example improvement. In (a), the L-graph of join node 5 has only O-node, node 3. Node 3 is duplicated into two clones $3A$ and $3B$, as shown in (b). A subgraph (the graph induced by nodes $3B$, 4, and 5) of the L-graph of node 5 in (b) is replaced by a single node X, as shown in (c). As a result, there is one O-node, node X, in the L-graph of node 10 in (c). Node X is duplicated into nodes XA and XB, as shown in (d). Node Y in (e) corresponds to the block of $3A$, XA, 6, 8, and 10. Finally, the L-graph of node 11 in (e) is reduced to a single node Z, as shown in (f). In this example, the binding of nodes 4 and 5 as well as node 3 should

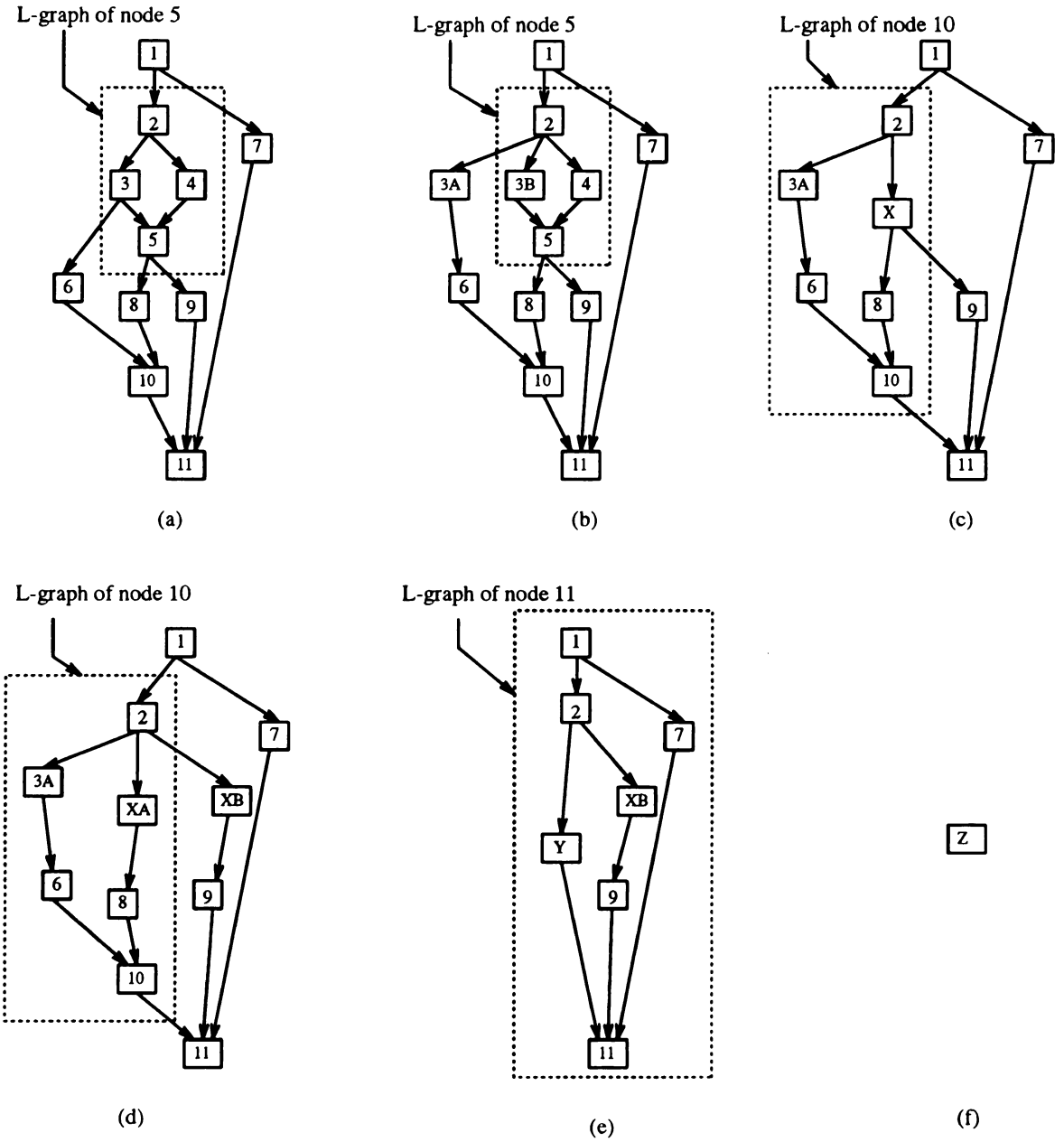


Figure 4.12. An improvement example

Algorithm *Improvement_by_Cloning* (G)
begin
Step 0. Find all nodes of G whose binding must be fixed.
Step 1. Visit each join-node v of G in topological order, and perform Step 2 and Step 3.
Step 2. Eliminate the O-nodes from the L-graph $G'(V', E')$ of v by cloning.
Step 3. Apply "the cell selection algorithm for a series-parallel graph" to the L-graph of v.
end *Improvement_by_Cloning*

Figure 4.13. Procedure *Improvement_by_Cloning*

be fixed since the node X, which includes 4 and 5, is cloned.

Theorem 2 asserts the correctness of the procedure in Figure 4.13.

Theorem 2: Let $G(V, E)$ be a two-terminal DAG such that the sink node of G is a join node. When the procedure of Figure 4.13 is applied to G, the following (i)-(iv) hold.

(i) Just before executing Step 2, there is only one L-graph of v.

(ii) After finishing Step 2, the resulting L-graph of v is a series-parallel graph.

(iii) After Step 3, the resulting L-graph of v produced by Step 2 is reduced to either a single node or a chain of two nodes.

(iv) After processing the last join node, which is the sink node of G, the resulting graph has only one node.

Proof:

(i): If v is the first join node of G in topological order, there is no predecessor of v which is a join node. Therefore, there is only L-graph of v by Lemma 2. Since we remove join nodes in topological order, when a node v is visited, there is no predecessor of v which is a join node. Thus, (i) holds.

(ii): When v is visited, there is no predecessor of v which is a join node. Note that

Step 2 eliminates O-nodes from the L-graph of v and does not introduce any new join node. Thus, (ii) holds from Lemma 3.

(iii): Immediately follows from the behavior of the cell selection algorithm of Section 4.2.2. Let w be the nearest common predecessor of all the immediate predecessors of v. If w has no out-going edge to a node not in L-graph of v, the L-graph is reduced to a single node. Otherwise, the L-graph of v except for w is reduced to a single node, implying that the L-graph is reduced to a two-node chain.

(iv): Trivially holds. □

4.5.3 Complexity Issues II

The number of nodes in G is $O(|V|)$ at any time during execution of *Improvement_by_Cloning*. For a join node v, the number of the nodes to be cloned is at most the number of nodes in the L-graph of v. Thus, each Step 2 can be done in $O(|V| + |E|)$. At Step 3, the resulting L-graph of v is reduced to either a single node or a chain of length 2, which can be done in pseudo-polynomial time. Since Step 2 and Step 3 must be repeated as many times as the number of the join nodes of G, the time complexity of *Improvement_by_Cloning* is pseudo polynomial.

4.6 Experimental Results

This section presents the experimental results of our algorithm with the LGSynth91 logic synthesis benchmark circuits. We compare our results with the outputs of misII, a technology mapping system from UC Berkeley, and Lin's work [57]. For each of the benchmark circuits, we first ran misII on it using an example technology library. The technology library used in this experimentation is a variation of a technology

library in the LGSynth91 logic synthesis benchmark package. Our algorithm is then tested using the technology library and the network of gates. In misII, there is a user-determined parameter called m. For $m = 0$, misII produces an area optimal solution while ignoring the maximum allowable delay. In contrast, for m=1, misII produces a solution with the longest delay less than the maximum allowable delay, but does not optimize the area of the output. We used m=0.75 to avoid those extreme cases and to compare with Lin's work in which m is around 0.75. For each circuit, T_{max} was set to the longest delay of the circuit obtained from misII.

Table 4.2 shows our results, produced before and after the second part of our algorithm (*i.e.*, improvement by cloning). The column *size* contains the number of gates in each circuit obtained from misII. The *cloning ratio* of a graph refers to the ratio of "the number of gates to be cloned" to "the total number of gates". For each circuit, the CPU time was measured on a SUN 4 Sparcstation. *No of iterations* is the number of iterations made from Steps 3 through 5. The outputs of Lin's work shown in this table are the area improvements over misII, reported in [57].

For almost all of the circuits, our algorithm achieves a significantly better area improvement over the outputs of misII than Lin's work. The area improvement over misII ranges between 27% and 59%. In all cases, our algorithm produces solutions in a reasonable time. The CPU time depends on T_{max} as well as the number of gates.

The run time of our algorithm depends on N, which is the number of significant digits of delay values. The results in table 4.2 are obtained when $N = 3$. Table 4.3 illustrates the comparison of the results for different N's, where A_1 and A_2 are the area of each circuit obtained for $N = 3$ and $N = 4$, respectively. As shown in the table, the area improvement gained by the more accurate delay representation is negligible.

The cloning method improves the initial solution further. The improvement is

Benchmark Circuits		misII			Algorithm					Lin's Work
Name	Function	Size	Area	Delay	Area improvement		No. of iterations	Cloning ratio	CPU time	Area improvement
					Before postprocessing	After postprocessing				
b9	Logic	86	4224	17.39	2618 (38%)	2399 (43%)	5	75%	1 sec	-0.2%
C432	Priority Decoder	149	7319	53.67	4903 (33%)	4353 (40%)	3	50%	1.5 sec	35%
C499	Error Correcting	310	15126	44.39	11193(26%)	10976 (27%)	5	70%	3 sec	29%
C880	ALU and Control	245	12150	41.20	6682(45%)	6040 (50%)	4	70%	2 sec	N/A
C1355	Error Correcting	470	23086	50.42	13389(42%)	12703 (44%)	6	90%	5 sec	N/A
C1908	Error Correcting	398	21126	58.70	12041(43%)	11229 (46%)	6	80%	5 sec	15%
C2670	ALU and Control	561	29018	59.75	16250(44%)	14990 (48%)	5	78%	6 sec	13%
C3540	ALU and Control	793	42039	77.38	16250(44%)	21810 (48%)	6	84%	8.5 sec	N/A
C5315	ALU and Selector	1305	68950	68.03	37233(46%)	36152 (47%)	5	90%	15 sec	N/A
C6288	16-bit Multiplier	2341	116938	181.45	47944(59%)	47230 (59%)	7	95%	30 sec	N/A
C7552	ALU and Control	1813	90571	97.38	50719(44%)	45910(49%)	8	80%	25 sec	20%
alu2	ALU	245	13073	69.62	8235(37%)	7474 (43%)	5	66%	3.5 sec	20%
alu4	ALU	463	24112	70.66	14949(38%)	13898 (42%)	7	75%	8.5 sec	16%
dalu	Delicated ALU	1197	62539	89.84	34396(45%)	33416 (47%)	8	64%	19 sec	16%

Table 4.2. A summary of experimental results (no. of significant digits = 3)

Circuit	misII	Our algorithm				
		No of significant digits=3		No of significant digits=4		Ratio
Name	Area	Area (A_1)	CPU Time	Area (A_2)	CPU Time	A_1/A_2
b9	4224	2399	1s	2380	3 sec	99%
C1908	21126	11229	5s	11208	20 sec	99%
C2670	29018	14990	6s	14900	22 sec	93%

Table 4.3. A comparison of runs with different precisions

more effective when the cloning ratio is low. The lower the cloning ratio of a circuit, the more gates will have optimal bindings.

4.7 Conclusion

In this chapter, we presented the time complexity of the cell selection problem and a path-oriented algorithm. We proved that the cell selection problem is strongly NP-complete for a general acyclic graph even when there are only two library cells for each gate. Our cell selection algorithm iteratively picks paths of gates and finds a new binding of the paths until the longest delay decreases to a user-defined upper bound. After finding a solution, the algorithm improves it by constructing a cloned series-parallel graph and then obtaining a new binding of the graph.

In the experimentation with the LGSynth91 logic synthesis benchmark circuits, the path-oriented feature of our algorithm is shown to be effective. On an average, the results of our algorithm improve the outputs of misII by 45%. The results are significantly better than those of Lin's work.

CHAPTER 5

Workspace Management in VLSI Design Processes

This chapter presents a methodology for workspace management in VLSI design processes. A workspace model is proposed which reflects the way that the entire process of the project is hierarchically sub-divided into subprocesses. Our proposed workspace model captures data dependencies between workspaces, created separately for subprocesses of a project. This feature facilitates change propagation, since designers can easily recognize whether their work is affected by design objects released by someone else. Under our workspace model, we present a mechanism for preserving and keeping consistent states of an in-progress project. In most of previous VLSI/CAD frameworks, keeping data consistencies among workspaces is left to designers. We present mechanisms for version control over the outputs of CAD tools, and their variations from previous work.

5.1 Introduction

The electronic design automation community has recently placed its emphasis on the *process* of design creation and evolution. A VLSI design process is defined as a sequence of tools and subprocesses which performs a CAD task [35], for example a high-level synthesis on a VHDL behavioral object. The execution of a VLSI design process is iterative and tentative [47], resulting in many versions. With a large design process, a design team hierarchically sub-divides it into small processes in terms of its target CAD function and its target design, and cooperatively works on them. Therefore, CAD frameworks [7, 47, 49, 77] support the concept of workspaces in order to manage design objects separately for subprocesses, and provide version control over design objects in workspaces. Workspaces can be defined as areas in which process-related design objects are placed [47, 77].

This chapter presents a workspace model and workspace management mechanisms for VLSI design processes. In the VLSI/CAD literature, litte research, except for [7, 47], have been reported on workspace modeling. The main contribution of the previous workspace models [7, 47] is to organize workspaces into a three-level hierarchy and define the concept of *release*. In the previous models, workspaces are named private (for the process performed by an individual designer), group (for the process performed by a group of designers), or project workspaces, depending on their levels in a hierarchy. Design objects are released (meaning "relocated") from a private (group) workspace to a group (project, respectively) workspace after they are validated and ready for others to use. The previous models are limited in sufficiently capturing how more than one designer cooperate to carry out a project. For example, when design objects are released by some designer, it is unclear which designers should be notified of them. It is also unclear how far design objects designed by designers should be released. That is, designers cannot easily recognize whether these objects should be

released into only a group workspace or into a project workspace.

One principle of our workspace model presented herein is that a workspace hierarchy should reflect the way that the entire process of a project is hierarchically partitioned into subprocesses. Workspaces are separately created for these subprocesses. Our model captures data dependencies between subprocesses, *i.e.*, input and output relationships between subprocesses, and can overcome the shortcomings of previous models. CAD frameworks provide various mechanisms for efficiently managing design objects in workspaces. They include mechanisms for the preserving of design states [7], change propagation among design objects [19, 27, 46], version control over design objects produced by tool invocations [12, 44, 40], and and other basic operations (such as check-in and check-out). The chapter also presents such mechanisms which are based on our workspace model. Few CAD frameworks, except for the DDM system [7], provide mechanisms for preserving of design states. However, the mechanism provided by the DDM system is minimal in that the framework can only records states of an individual workspace. Our mechanism for preserving design states allows designers to preserve states of an entire project, compare two different states for seeing design tradeoffs, and restore a workspace (or a entire project) to a preserved state.

Change propagation is system support for triggering proper actions against changes made to design objects. This system support serves as a vehicle for keeping data consistencies among workspaces. The major difference between our change propagation method and previous work [19, 27, 46] is that in our method, changes are reported to workspaces, not designers as in the previous work. This feature enables designers to easily identify the changes affecting a particular workspace, when a designer is working in more than one workspace. In the literature, several CAD frameworks, notably Ulysses [13, 14], Cadweld [30], NELSIS-CAD [12], Monitor [44],

Tzi-cker's system [18], and Minerva [43], have been developed which provide design process management. These frameworks are primarily aimed at executing a user-specified sequence of CAD tools, and provide a limited capability of data management for tool executions. Their only capability is versioning to distinguish an old output and a new output of the same tool when the tool is executed more than once. This capability alone is not sufficient for meeting various designers' needs, such as capturing a design history or preventing the proliferation of temporary data resulting from "what if"-style design. Lastly, this chapter also presents basic mechanisms and their variations from previous work.

This chapter is organized as follows. Section 5.2 describes structuring concepts of our workspace model. Section 5.3 describes a mechanism for preserving design states and using them in the design process. Section 5.4 shows how efficiently change propagation are handled in the workspace model. Section 5.5 describes techniques for version control over outputs of tool executions. Section 5.6 presents basic workspace management mechanisms. Section 5.7 concludes.

5.2 A Workspace Model

5.2.1 Modeling Concepts I

Workspaces provide an environment for carrying out tasks in VLSI/CAD frameworks and CASE (Computer-Aided Software Engineering) systems. In CAD frameworks and CASE systems, workspaces are used in two ways. One is that workspaces act as a medium for communication between designers [7]. That is, designers release design objects into a workspace, from which other designers access the objects for their work. The other is that workspaces are a place to which a particular design

process (or a designer) is allowed read/write privilege [27, 47, 77]. This implies that a workspace corresponds to a directory of the UNIX file system, in which files are created and updated. In our workspace model, workspaces for a project are hierarchically organized, as illustrated in Figure 5.1. Workspaces in a workspace hierarchy are classified into two categories, depending on how they are used.

The two categories are:

- *Local workspace*: A leaf node, *i.e.*, a workspace with no child workspace, is a local workspace. Local workspaces are ones in which CAD tools are executed to accomplish tasks.

- *Shared workspace*: A non leaf node is a shared workspace. A shared workspace is one which contains design objects released from its child workspaces.

After a design object is initially created, new versions can be derived from it by modifying it or by changing its constraints, and new versions can in turn be derived from them. In this chapter, we will call all of these versions, including the object itself, a *version group* for the object*. The major role of workspaces is to specify which version is the currently used one. Therefore, workspaces, local or shared, contain only *one* version in a version group. A shared workspace corresponds to a group workspace in the terminology of previous workspace models [7, 47, 52]. A workspace and its parent workspace can have different versions in the same version group, as in the previous workspace models. The content of a shared workspace is *inherited* by all descendants of the workspace, as in [7]. This inheritance means that for a version group G for an initial object D, if a local workspace W does not contain a version in G, a reference to D made within W is resolved by any version in G within the *nearest* ancestor

*Versions in a version group for an inital object have same name as the object but different version numbers.

workspace W. With a design object, workspaces contain either an actual copy of it or a reference (or a pointer) to it. Actual copies of design objects can be modified in a workspace, but references to design objects denote that the design objects are read-only. Actual copies of design objects pointed by references are stored in a design library† References are specialized into two types, static and dynamic, depending on whether design objects pointed by them change over time. Dynamic references point to the latest versions as new versions are checked in a design library‡ They are useful for enabling designers to automatically use up-to-date design objects.

---

†A design library is an archive of design objects, which are created and modified within a workspace.

‡In CAD frameworks and CASE systems, the operation *check-in* is the registration of a version into a design library from a workspace. An operation complimentary to check-in is called *check-out*, which adds a version (which is either an actual copy or a reference), which has been registered in a design library, to a workspace. Section 5.6 will describe more detail about the two operations.

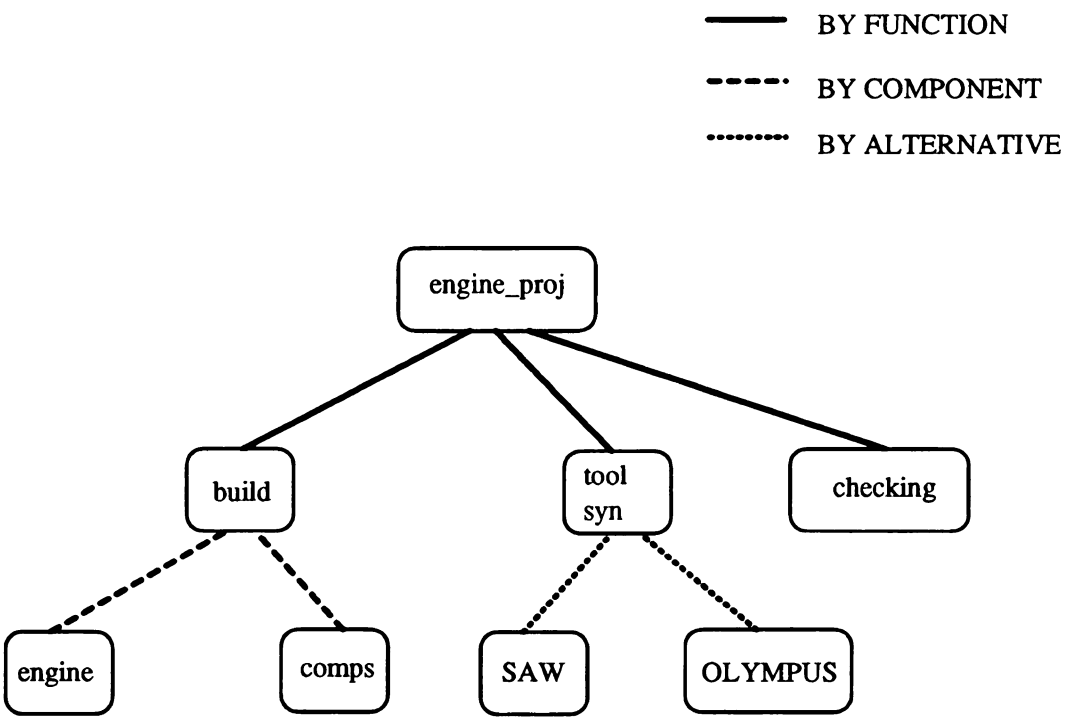


Figure 5.1. Three Dimensions of Workspace Hierarchy Construction

A workspace hierarchy usually exists for a project. The entire design process (*i.e.*, work) of a large project is usually hierarchically partitioned into small processes, which are cooperatively carried out by a design team. A basic principle underlying our workspace model is that a workspace is created for each process at a level. A workspace hierarchy is constructed along three orthogonal directions: *by-component, by-function*, and *by-alternative*. These three directions are the ways of decomposing an entire design process. With a project, if the target design of the project is large, the design can be hierarchically partitioned into components. Then different workspaces are created for all the components (by-component direction). If the target function of the project is large, the function can be hierarchically decomposed into small functions. Small functions are performed in different workspaces (by-function

direction). In performing a task, a designer tends to try alternative ways of designing (called *design alternatives*) and evaluate their tradeoffs. Different workspaces are created for those alternative ways (by-alternative). Thus, a workspace hierarchy consists of workspaces at various levels. A workspace at any level can have child workspaces along any of the three directions.

Note that with a non-leaf workspace for a design process P, child workspaces are created for P's subprocesses. Therefore, we can easily obtain data dependencies among the child workspaces. Data dependencies among workspaces are input-and-output dependencies between P's subprocesses. These dependencies are useful for designers who are cooperatively carrying out the process P. Such designers will communicate with each other by releasing and use design objects through the non-leaf workspace. From these dependencies among the child workspaces, they can easily recognize whether the new release of design objects from a child workspace affects their work. Our data model allows designers to specify data dependencies among child workspaces.

Consider a project that is aimed at synthesizing an RTL-level structural description of an engine control unit. The entire process of the project is first sub-divided into three subprocesses, which are for writing a VHDL model, **engine.vhd**, of the control unit, executing high-level synthesis tools with the VHDL model, and checking whether the VHDL model matches a synthesized, structural description, respectively. Since the VHDL model **engine.vhd** is large, the process for writing it is further sub-divided into two subprocesses, which are for writing components and building an entire design from the components, respectively. To see tradeoffs between two high-level synthesis tools, called the **SAW** system [72] and the **OLYMPUS** system [53], the process for synthesis is further sub-divided into two subprocesses for the two tools. A workspace hierarchy of the project shown in Figure 5.1. Solid, dashed, and

dotted lines denote by-function, by-component, and by-alternative decompositions, respectively.

The top level workspace, `engine_proj`, is for the whole project. Components of the VHDL model `engine.vhd` are designed in workspace `comps`. They are released into the workspace `build`, from which the workspace `engine` inherits them. Once the entire VHDL model is built in the workspace `engine`, it is released into the workspace `build`. After the components and the VHDL model in the workspace `build` are validated by designers working in `engine` and `comps`, they are released into the workspace `engine_proj`. Then, they become visible to designers working in the workspaces `SAW` and `OLYMPUS`. Once two structural descriptions are released from the workspaces `SAW` and `OLYMPUS` into the workspace `tool syn`, they are evaluated. Among them, a better one, for example one with smaller area, is selected and released into the workspace `engine_proj`. The workspace `checking` is used for the checking process.

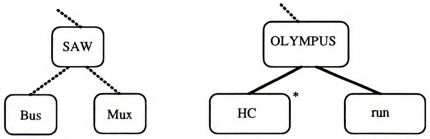


Figure 5.2. Workspace Hierarchy Changing

5.2.2 Modeling Concepts II

A workspace hierarchy is initially constructed in the beginning of a project. Leaf-node workspaces are assigned to designers. Our workspace model allows a designer to have more than one leaf-node workspace. It is assumed in previous workspace models [7, 27, 47] that each designer can have only one leaf-node (private workspace in their terminology) workspace. This assumption is inadequate for a situation when a designer is involved in two or more different design processes. Then, the private workspace for such a designer will contain the objects produced by these design processes.

A design process operating on a leaf node may need to be partitioned further into subprocesses during the project. To handle this, our workspace model allows a workspace hierarchy to be dynamically changed in a way that workspaces for new subprocesses are created, and added as new leaf nodes. For example, Figure 5.2 shows new workspaces which are added to the workspace hierarchy of Figure 5.1. The new workspaces denote the following. Using the `SAW` system, a designer explores two design styles, mux-based and bus-based design, in workspaces `Mux` and `Bus`, respectively, as alternatives. A designer, working in the workspace `OLYMPUS`, partitions his entire work into two subprocesses, and creates two workspaces. One workspace, `HC`, is for writing an HC (Hardware-C) model from the VHDL model in the workspace `engine_proj`[§] The other workspace, `run`, is for actually running the OLYMPUS system.

When a process is sub-divided into subprocesses, if a subprocess produces design objects that are used only within the process, the design objects need not be accessible from other processes. It is enough to release the design objects into the workspace for the process. That is, they need not be released further along the workspace hierarchy.

[§]The OLYMPUS system accepts a Hardware-C behavioral description.

Our workspace model distinguishes such a subprocess from others by marking the workspace for it with a star ('*'). An example of such a subprocess is in Figure 5.2, where an HC model is used only as an input to the OLYMPUS system.

The way of releasing design objects in our workspace hierarchy is as follows. Designers work in local workspaces until they have design objects ready for others to use. Then, the design objects are released into a parent, shared workspace, and held until approval is received. After approval, all of them, except for ones released from star-marked child workspaces, can be released further into an upper-level, shared workspace. This approval-and-release is repeated until they are in the top workspace.

A workspace hierarchy in previous workspace models are adequate only when an entire process is partitioned along the by-function or by-component direction. Instead, our workspace hierarchy can capture the structure of the search space, since design alternatives can be explored in separate workspaces. If a design alternative, say A, is sub-divided into more specific alternatives, separate workspaces for these alternatives are created as children of the workspace for alternative A. For example, in Figure 5.2, two alternatives, bus-based and mux-based design, are explored for high-level synthesis using the SAW system. This feature of our workspace model enables designers to easily find what design alternatives were explored, and how they are related within an entire design process.

5.3 Snapshots as a Mechanism for Preserving Design States

At any design point, the state of a project can be described by the design objects of the project [60]. Keeping design states is important for many reasons. For example,

a design team can restore the state of a project to some point in the past. Also, preserved states of a project can be used as checkpoints (review points) of the project.

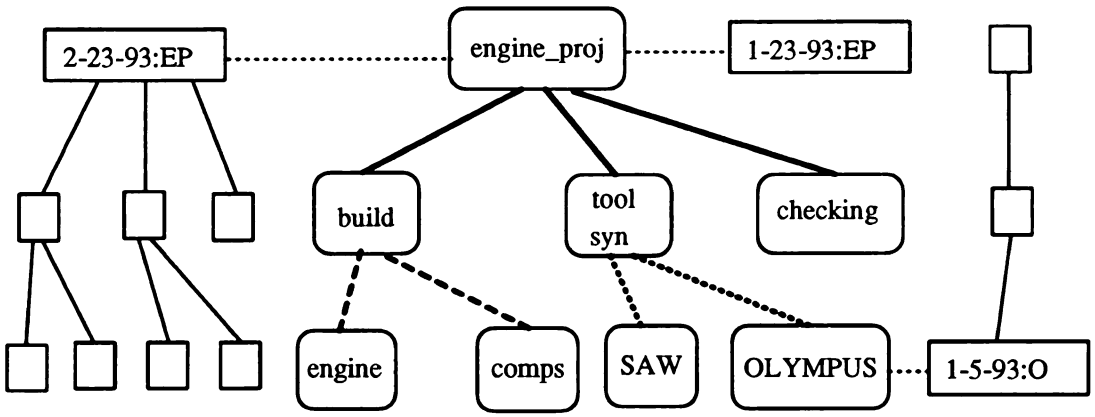


Figure 5.3. Snapshot Examples

Our workspace model provides the notion of "snapshot" as a mechanism for preserving design states. Our workspace model supports three ways in which snapshots are taken from a workspace hierarchy. Snapshots are called *simple*, *top-down hierarchical*, or *bottom-up hierarchical*, according to their way of construction. A simple snapshot is a collection of design objects which are present in a workspace at a certain point. Since the contents of workspaces, local or shared, vary with time, simple snapshots can be regarded as permanent records of workspaces. They contain static references to design objects. Top-down hierarchical snapshots are possible for only shared workspaces, and bottom-up ones are for only local workspaces. A top-down hierarchical snapshot of a shared workspace W is a simple snapshot of W together with simple snapshots of all W's *descendant* workspaces. Since workspaces are created for portions (or the entire work) of a project, this snapshot of a workspace captures

an *overall* state of the portion of a project for which the workspace is created. A bottom-up hierarchical snapshot of a local workspace W is a simple snapshot of W together with simple snapshots of all W's *ancestor* workspaces. Since design objects are inherited, this snapshot of a workspace captures all design objects that affect a design process operating on the workspace.

Figure 5.3 illustrates three example snapshots. Node `1-23-93:EP` (a snapshot made on January 23, 1993) is a simple snapshot of workspace `engine_proj`. A top-down hierarchical snapshot of workspace `engine_proj` is shown, which is made up of the node `1-23-93:EP` and all its descendant nodes. A bottom-up hierarchical snapshot of workspace `OLYMPUS` consists of the node `1-5-93:O` and all its ancestor nodes.

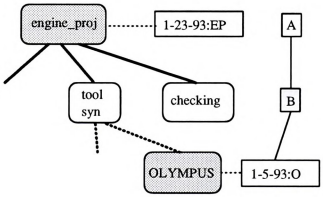


Figure 5.4. Use of snapshots

When designers have lost data consistencies among objects in workspaces, they will want to backtrack to a previous, consistent state. In our workspace model, a workspace can return to a previous design state by loading a snapshot, simple or

hierarchical, which preserves the design state. For example, in Figure 5.4, workspace `engine_proj` can return to the design state of January 23, 1993 by loading simple snapshot `1-23-93:EP`. A special consideration is necessary for loading a local workspace with a bottom-up, hierarchical snapshot. Consider workspace `OLYMPUS` and a bottom-up, hierarchical snapshot in Figure 5.4. Workspace `OLYMPUS` must be filled up with all design objects accessible from simple snapshot `1-5-93`. Such objects are all objects in simple snapshot `1-5-93` together with all objects inherited from simple snapshots A and B. If two versions in the same version group appear in snapshots A and B, respectively, only the version in snapshot B is inherited by workspace `OLYMPUS`.

By comparing preserved states of a project, designers can obtain information about how differently design processes were performed at the states. Such information is useful for designers which want to understand why different sets of design objects were produced at design states. Our workspace management methodology includes a mechanism for comparing snapshots and identifying differences between their contents. Comparison between two simple snapshots produces two lists of versions. One is a list of design objects that are in only one of the two snapshots. The other is a list of pairs of design objects D_1 and D_2 such that D_1 is in one snapshot, D_2 is in the other snapshot, and D_1 and D_2 are different versions in the same version group. Comparison between two hierarchical (bottom-up or top-down) snapshots is carried out by comparing each pair of corresponding nodes (which are simple snapshots).

5.4 Change Propagation among Design Objects

We define change propagation as system support for monitoring changes to design objects and taking actions for keeping consistencies among design objects [19, 47]. In

our workspace model, changes within a local workspace (*i.e.*, creation or modification of design objects) are not seen from any other workspace. Therefore, changes are propagated to other workspaces only if they are from the following four categories. (1) checking in a design object (into the design library), (2) destroying a design object from the design library, (3) releasing design objects into a shared workspace, and (4) removing design objects from a shared workspace.

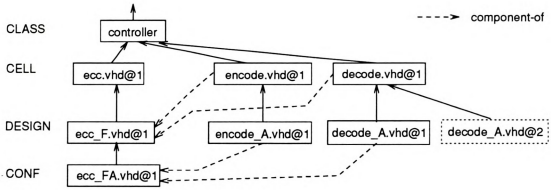


Figure 5.5. An Example of Automatic Configuration Binding

5.4.1 Changes to Design Objects in the Design Library

This subsection describes our change propagation method for changes from category 1 and 2. For a dynamic reference pointing to a design object, if a new version of the object is checked-in (category 1), the reference is adjusted so it points to the new version. The appearance of a new version can be propagated further. Our workspace management methodology includes a mechanism which automatically performs configuration binding on a DESIGN object, say O, when a new version of a component

of O is created. This mechanism is useful for a circuit in which various versions of O's components are under design. A designer need not explicitly obtain a complete design from O and a new version of its component. To support this mechanism, DESIGN objects have a reserved, boolean attribute, called *dynamic_flag*, which specifies whether configuration binding must be triggered. More specifically, automatic configuration binding on a DESIGN object is performed in a way that, if more than one design object is a candidate for a component, the newest one is selected and bound to the component. The reason for selecting the newest one is that this mechanism is primarily aimed at helping designers integrate a newly created version of a component into the entire design. For example, consider a DESIGN object `ecc_F.vhd@1` (a signal encoding-decoding circuit) with *dynamic_flag* being *true*, shown in Figure 5.5. As illustrated in the figure, the DESIGN object has two components, which are instances of CELL objects `encode.vhd@1` (a signal encoder) and `decode.vhd@1` (a signal decoder). A CONF object `ecc_FA.vhd@1` was automatically derived from `ecc_F.vhd@1`, when DESIGN objects `encode_A.vhd@1` and `decode_A.vhd@1`, implementations of the two components of `ecc_F.vhd@1`, were created. If a new version, `decode_A.vhd@2`, is created as a child of `decode.vhd@1`, the version, together with `encode_A.vhd@1`, will lead to a new version, say `ecc_FA.vhd@2`, of `ecc_FA.vhd`. If dynamic references to `ecc_FA.vhd@1` are currently in a workspace, they will be updated to point to `ecc_FA.vhd@2`. Our mechanism for automatic configuration binding corresponds to the notion of change propagation along the component hierarchy, found in previous work [47]; when a new version of a component of a configuration is created, a new version of the configuration is automatically created by replacing an old version of the component with the new one. In addition to the creation of a new configuration, our mechanism allows designers to limit the scope of change propagation, a feature not supported by the notion. By imposing constraints on configuration binding, a designer can obtain only those configurations of interest. For example, when DESIGN

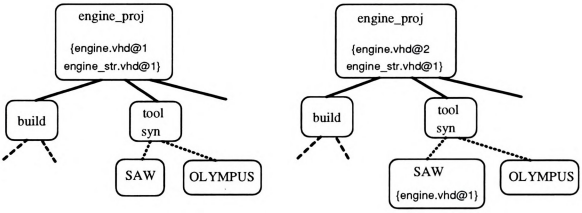
object `ecc_F.vhd@1` in Figure 5.5 has a constraint that restricts the total size to 1000, configurations with total size over 1000 will not be derived from `ecc_F.vhd@1`.

Next let us consider changes from category 2. If a design object is to be deleted, the deletion is allowed only when it (either an actual copy of it or a reference to it) is not in any workspace, and it is used not as a component of other design objects. This restriction is necessary to keep all design data in the design library and workspaces complete.

5.4.2 Changes to Design Objects Released in a Workspace

This subsection describes change propagation for changes from category 3 and 4. The release of versions tends to be *iterative* in the following sense: various versions are sequentially released into a shared workspace from a child workspace until the latest released is approved by the designers who use it. There can be a situation in which designers want to postpone the effect of a new release on their in-progress work, and to keep using the previous version. This situation may occur when a long-duration task is in progress. For example, consider a designer, working in workspace `SAW` of Figure 5.1, who knows little of how to use the SAW system. Suppose that the designer is just learning the SAW system with a VHDL model inherited from workspace `engine_proj`. When a new version of the VHDL model is released, the designer will not use it immediately, since the designer needs to understand it first.

Our workspace management methodology includes a mechanism for enabling a designer to keep using the latest released versions. Design objects in shared workspaces are associated with an attribute called *static-inheritance*. The value of this attribute is a list of local workspaces. We will use an example to explain this attribute. Suppose that a design object D is in a shared workspace, and that the value of its static-



(a) Before release of engine.vhd@2

(b) After release of engine.vhd@2

Figure 5.6. An Example of a New Release

inheritance attribute is a list of two workspaces W_1 and W_2. When a new version V of D is released into the shared workspace by a designer, the object D is automatically copied into workspaces W_1 and W_2 prior to the release. In our workspace management methodology, the value of its static-inheritance attribute is updated by designers.

Designers will remove an object from a shared workspace if they would find it inappropriate for the release into the workspace. Also, when a design object is in a workspace, if a new version of the object is released into the workspace, the release will lead to the removal of that object. When a design object is removed from a shared workspace, all designers working in descendant, local workspaces of the shared workspace should recognize the removal. Then they will take proper actions against the removal.

Changes from categories 3 and 4 may cause objects in shared workspaces to be

invalid. Consider two design objects, say D_1 and D_2, in shared workspaces such that D_1 has been used in the design of D_2. If D_1 is removed because of a change from category 3 or 4, D_2 will become invalid. In our workspace management methodology, design objects in a shared workspace have an attribute, called *use*, for keeping "used" relationships between two objects. For an object, the value of its use attribute is a list of design objects in shared workspaces that are used in the design of it. The value of the use attribute is given by a designer, enabling the attribute to capture any application-specific "used" relationship, including "component-of" (one is a component of the other) and "produced-from" (one was used as an input of a CAD tool which has produced the other), between two objects. If a change from category 3 or 4 is made to a design object D in a shared workspace, all design objects in a shared workspace which have D in their use attribute, are marked "invalid." Also, if design objects in shared workspaces have any "invalid"-marked object in their use attribute, they are also marked "invalid". This marking prevents designers from accessing invalid objects, helping them maintain data consistencies among released objects.

For example, consider Figure 5.6 (a), where a released object `engine_str.vhd@1` (an RTL-level structural description synthesized from `engine.vhd@1`) is in the use attribute of a released object `engine.vhd@1`. Workspace `SAW` is in the static-inheritance attribute of the object `engine.vhd@1`. When a new version, `engine.vhd@2`, is released into workspace `engine_proj`, the previous version (*i.e.,* `engine.vhd@2`) is copied into workspace `SAW`, as shown in Figure 5.6 (b). Also, the released object `engine_str.vhd@1` is marked "invalid."

5.4.3 Change Notification

Since CAD frameworks can hardly understand the full effects of changes, designers are eventually responsible for responses to changes. Our workspace management methodology includes a mechanism for notifying relevant designers of changes, as in [19, 27, 46]. Changes are reported directly to designers in previous work, but this method has a limitation. If a designer works in more than one workspace, the designer should identify which changes are related to each of his workspaces. To relieve designers from this burden, our mechanism for change notification is workspace-based. In other words, changes are reported to *workspaces* affected by them.

The registration of new versions into the design library, changes from category 1, is reported to all workspaces with their old versions checked-in. Releasing new versions or deleting versions from a shared workspace, changes from category 3 or 4, are reported to all descendant, local workspaces of the shared workspace.

Our workspace management methodology also includes a mechanism which queries a workspace and retrieves those of interest among all new versions reported to the workspace. A query is a set of conditions, which are boolean predicates defined on attributes of design objects. Each condition is in the form of A=V (*e.g.*, `author=John`), where A is an attribute, and V a value. The value A can be one of two reserved strings, `up` and `down`, if attribute A is one that measures an electrical or geometrical property, such as size, delay, or power. Consider an example condition `size=down`. Among versions reported to a workspace, this condition is true for only those with size smaller than their old versions contained in the workspace. When a designer is optimizing the area of a design, the designer will focus on only new versions of components with size smaller than those in the workspace.

5.5 Version Control in Local Workspaces

A VLSI design process is executed by invoking CAD tools in a local workspace. As the process is iteratively executed, it will result in various versions. Version control in a local workspace should be *process-oriented* , in that it is automatically exerted during a process execution. Process-oriented version control is not a well-defined concept in the CAD framework literature. This section defines essential mechanisms that, we believe, need to be supported for this style of version control.

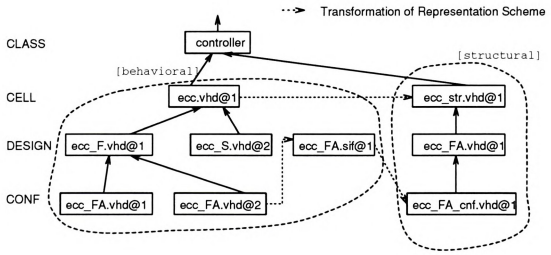


Figure 5.7. An Example Design Library

(1) Design Objects Produced by Tool Invocations

A local workspace contains not only design objects checked in, but also tool outputs. A *tool output* is a design object which is produced from another object (either an object checked in or an object produced from it using a CAD tool), called a *tool*

input, by invoking a CAD tool. Version control in local workspaces needs to support system-defined attributes of a tool output which store information about its tool invocation. Such attributes are necessary to capture dependencies between design objects [18]. They include the name of a tool, the name of a tool input, and a list of tool parameters. Values of these attributes can be given by an application, called the process manager [12, 18, 44], which actually invokes tools for a CAD task, or a designer. One type of a CAD tool is an editor, including a schematic editor and a LAYOut editor. Note that designers edit a design object and produces a version of it for various reasons. Example reasons are bug fixes, structural changes, functional refinement, addition of comments, etc. We support an attribute of design objects which specifies the reason for editing. Values of the attribute must be given by designers. This attribute will help designers document design processes for future reference.

Once a tool output is produced, it may not belong to another one as a child in the abstraction hierarchy of the design library. Such an object is usually one in a representation scheme (*e.g.*, schematic, netlist, etc.) different from its tool input. The data model of chapter 2 is adjusted to allow design objects without parents. Consider an example of design library in Figure 5.7. In the figure, a data control flow graph `ecc_FA.sif@1`, produced from a VHDL behavioral `ecc_FA.vhd@2`, has no abstraction relationship to any other.

(2) **Versioning of Tool Outputs**

Tool outputs need to be versioned as the same tool is repeatedly executed. With most CAD frameworks [7, 11, 12, 18, 44], all versions of tool outputs are kept within local workspaces while tasks are being performed. This style has a shortcoming in that design objects in local workspaces are not accessible from other local workspaces unless designers explicitly move it to the design library. One policy of our version control is that all versions of tool outputs, except for ones being currently used, are

kept in the design library. This will ease cooperation between designers who frequently exchange data during their process execution.

(3) **Control over Proliferation of Versions**

Versions of design objects are produced during process executions. Sometimes some of them are intermediate results, so it is better to not store them in the design library. To solve this problem, each design object in a local workspace is associated with a boolean flag, called *version flag*. Once a tool has produced a design object `D`, the object is kept in only a workspace until it is registered into the design library. Suppose that a new version `D'` of the object is about to be placed in the workspace. The registration of `D` into the design library occurs only when the version flag of `D` is *true*. However, if the version flag of `D` is *false*, V is simply overwritten with V'. This prohibits the proliferation of intermediate versions.

5.6 Workspace Operations

In this section, we describe basic operations needed for workspace management, and their variations from those in the previous CAD frameworks.

- **Check-out:** Check-out is an operation that retrieves a design object into a local workspace from the design library. Unlike most CAD frameworks [7, 47, 60, 77], designers can check out either an actual copy of it or a reference to it. There is no distinction between *read-only* and *read-write* check-out. Each actual copy of a checked-out object is modifiable, and a modified copy is returned (*i.e.*, checked-in) to the design library as a new version. Check-out is performed in one of three ways: *simple*, *component hierarchical*, and *abstraction hierarchical*. A simple check-out retrieves an object in isolation. Component hierarchical

check-out checks out a CONF object and all its component objects. abstraction hierarchical check-out, not found in previous CAD frameworks, checks out a design object and all its ancestor objects, excluding CLASS objects. Note that CLASS objects are used as groups of similar design objects, so they are not actual data.

- **Check-in:** Check-in is an operation that registers a design object from workspaces into the design library. As opposed to standard check-in [57, 66, 73], a design object to be checked in can be one which has not been checked out earlier. This check-in is necessary for tool outputs produced during a process execution. Check-in of new versions will trigger change propagation of category 1, as mentioned in Section 5.4.1.

- **Release-in/Release-back:** Operation release-in releases a set of versions from a workspace into the parent of the workspace unless the workspace is star-marked. No corresponding concept is found in the literature. Releasing a new set of versions will trigger change propagation of category 3. Operation release-back deletes a version from a workspace. If a version is deleted from a shared workspace, change propagation of category 4 is triggered.

- **Workspace-Create/Workspace-Remove:** These operations are applied to only leaf nodes of a workspace hierarchy. Workspace-create is an operation which creates either a child of a node in a workspace hierarchy or the root node of a workspace hierarchy. Workspace-remove is an operation which deletes a leaf node in a workspace hierarchy.

5.7 Conclusion

We presented a methodology for workspace management. The methodology is based on a workspace model which organizes workspaces to reflect the way that the entire work of a project is performed by a design team. Since our workspace model captures data dependencies among workspaces, the model was shown to facilitate change propagation and notification. Our change notification is workspace-based in that changes are reported to affected workspaces. They feature prevents designers from unconsciously working with out-of-date design objects. We presented the notion of the snapshot as a mechanism for preserving and restoring consistent design states. Little research has been published on such a mechanism in the VLSI/CAD literature. We also presented mechanisms required for version control over tool outputs, and their variations from previous work.

CHAPTER 6

Conclusion and Future Research

VLSI design is characterized by a large volume of data, with diverse modalities, and team design in which a large design is sub-divided into small objects. In this thesis, we studied how both the object-oriented paradigm and constraints enforced on designs can be exploited to increase the modeling power of a VLSI data model, capture various dimensions of design evolution, and facilitate system support for configuration management. Also, we investigated how efficiently the path-oriented nature of cell selection, combined with node cloning, enables designers to obtain an area-minimum binding of a design under a maximum allowable delay.

6.1 Summary and Contributions of This Thesis

The contributions of this thesis are new techniques for version data modeling and configuration management in VLSI/CAD design environments. Major features of those techniques can be summarized as follows.

- An object-oriented VLSI data model was presented which organizes design objects hierarchically according to both their levels of abstraction and design constraints. This data model is superior to previous data models in that it not only captures various aspects of the VLSI design process, including stepwise refinement and constraint-driven design, but also efficiently supports a technology-independent methodology.

- A methodology for version modeling was presented which utilizes the concept of "level of abstraction" and constraints assigned to design objects. The version model was shown to provide a richer set of modeling concepts necessary for defining version relationships than previous work. Among those concepts, hidden versions and incompletely bound configurations increase flexibility in configuration binding. Relationships between versions were formally defined using design constraints. This feature not only provides a conceptual framework for automatically identifying relationships between versions, but also helps designers to explore versions of configurations. For example, versions of a configuration can be created by replacing components of a configuration with other objects with a certain relationship, such as replaceability, to the components.

- Little research has been performed on VLSI/CAD design environments which can provide control over configuration binding and ensure the correctness of configurations. A constraint-driven methodology for configuration binding was presented, which uses user-specified constraints in cell selection. This approach was shown to enforce selection of design objects with certain design styles or parameters preferred by designers, enhancing user interaction with VLSI/CAD design environments. Also it can detect design errors to validate the correctness of designs at early stages.

- An algorithm for cell selection was presented which combines iterative improvement with a cloning method. The algorithm finds the first solution by selecting paths from a given graph (a network of gates), and then obtaining a new binding of gates on the paths at each iteration. The first solution is improved further by obtaining a new binding of a cloned series-parallel graph. Our algorithm was shown to achieve significantly better performance than previous work. We proved that the cell selection problem is strongly NP-complete when a circuit is a general graph. Thus, finding a polynomial time algorithm for the cell selection problem in a general graph is not possible unless P=NP.

- A workspace model and workspace management mechanisms were presented. Few workspace models have been shown which can properly reflect on features of team design. Our workspace model hierarchically organizes workspaces based on the way that a project is being performed. Since the model captures how the design process of a project works, it facilitates release control and change propagation. We introduced the notion of snapshots as a vehicle for preserving design states of an in-progress project. We also presented mechanisms for allowing designers to shield themselves from changes made by others and continuing their work at consistent states.

6.2 Future Research

- Previous cell selection algorithms, including the algorithm presented herein, are applicable only to designs with a single level component. However, the description of a large design is usually hierarchical for decreased complexity. To our knowledge, no algorithm for the cell selection problem has been proposed for designs with a component hierarchy of more than one level. With a large design

with a deep component hierarchy, a flattened design of it, obtained from collapsing its component hierarchy, may have a very large number of components. Hence, previous algorithms do not efficiently work on such a flattened design. An algorithm needs to be developed which performs cell selection hierarchically, top-down, bottom-up, or a mixture, along the component hierarchy.

- Hardware engineers require version and configuration management provided by CASE tools, since they handle general files, such as manuals and progress reports. They also need to manage files generated through the hardware development life-cycle, which consists of various stages, such as specification, design, delivery, and maintenance. Present VLSI/CAD frameworks place emphasis on version and configuration management for data produced at only the design stage. For this reason, it is challenging to combine the techniques presented herein and CASE techniques to provide version control and configuration management at various stages of the hardware development life cycle in a seamless way.

- A large project may involve many designers, located at different sites, and many platforms. Especially, the design data of the project results from diverse tasks which are performed in the concurrent engineering style. To support workspace management for such a large project, the techniques presented herein should be extended to be applicable to a scenario where a team of designers cooperate in a distributed, concurrent engineering environment. In such an environment, for example, multiple check-outs can be performed on one design object from workspaces at remote sites. Our workspace model needs to be extended for efficiently handling change propagation and release control on workspaces which are geographically distributed.

- CASE systems for version and configuration management [73, 66, 27] employ various techniques for storing versions efficiently. Their basic idea is to store only differences, called deltas, between two successive versions of an ASCII file. Since CAD applications deal with many non-ascii files as well, storing only deltas is not suitable for VLSI/CAD data. Little work has been published on mechanisms for efficient storage of versions in VLSI/CAD data management. Such mechanisms will be format-specific to increase storage savings.

- Little research has been performed on a formal model of configurations. If various approaches to a configuration management mechanism, *e.g.*, configuration binding, are described using such a model, we can compare them and recognize their differences more easily. Moreover, the formal model will clarify what and how configuration management mechanisms need to be supported by VLSI design environments.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Acosta, R. D., Allexandre, M., Imken, G., and Read, B., "The Role of VHDL in the MCC CAD system," in *ACM/IEEE Design Automation Conference*, pp. 34 - 39, 1988.

[2] Adams, E. W., Honda, M., and Miller, T. C., "Object Management in a CASE Environment," in *International Conference on Software Engineering*, pp. 154 - 163, 1989.

[3] Afsarmanesh, H., McLeod, D., Knapp, D., and Parker, A., "An Extensible Object-Oriented Approach to Database for VLSI/CAD," in *International Conference on Very Large Database*, pp. 13 - 24, 1985.

[4] Armstrong, J., Cho. C., Shah, S., and Kosaraju, C., "The VHDL Validation Suite," in *ACM/IEEE Design Automation Conference*, pp. 2 - 4, 1990.

[5] Asano, T., "An Application of Duality to Edge-Deletion Problems," *SIAM Journal on Computing*, vol. 16, pp. 312 - 331, Apr. 1987.

[6] Baalbergen, E. H., Verstoep, K., and Tanenbaum, A. S., "On the design of the Amoeba Configuration Manager," in *International Workshop on Software Version and Configuration Control*, pp. 15 - 22, 1989.

[7] Banks, S., Bunting, C., Edwards, R., Fleming, L., and Hackett, P., "A Configuration Management System in a Data Management Framework," in *ACM/IEEE Design Automation Conference*, pp. 699 - 703, 1991.

[8] Barth, R. and Serlet, R., "A Structural Representation for VLSI Design," in *ACM/IEEE Design Automation Conference*, pp. 237 - 242, 1988.

[9] Batory, D. S. and Kim, W., "Modeling Concepts for VLSI Objects," *ACM Transaction on Database System*, vol. 10, pp. 289 - 321, Sept. 1985.

[10] D. Beech and B. Mahbod, "Generalized Version Control in an Object-Oriented Database," in *International Conference on Data Engineering*, pp. 14 - 22, 1988.

[11] Bingley, P., Bosch, O. ten, and Wolf, P. van der, "Incorporating Design Flow Management in a Framework based CAD System," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 538 - 545, 1992.

[12] Bosch, O. ten, Bingley, P., and Wolf, P. van der, "Design Flow Management in the NELSIS CAD Framework," in *ACM/IEEE Design Automation Conference*, pp. 711 - 716, 1991.

[13] Bushnell, M. L. and Director, S. W., "VLSI CAD Tool Integration Using the Ulysses Environment," in *ACM/IEEE Design Automation Conference*, pp. 55 - 61, 1986.

[14] Bushnell, M. and Director, S. W., "Automated Design Tool Execution in the Ulysses Design Environment," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 279 - 287, Mar. 1988.

[15] Casotto, A., Newton, A. R., and Sangio-Vincentelli, A., "Design Management Based on Design Traces," in *ACM/IEEE Design Automation Conference*, pp. 136 - 141, 1990.

[16] Chan, P. K., "Algorithms for Library-Specific Sizings of Combinational Logic," in *ACM/IEEE Design Automation Conference*, pp. 353 - 356, 1990.

[17] Chang, C-L, and Lee, R. C-T, *Symbolic Reasoning and Mechanical Reasoning.* Academic Press, New Work, 1973.

[18] Chieuh, T.-c. and Katz, R. H., "A History Model for Managing the VLSI Design Process," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 358 - 361, 1990.

[19] Chou, H. T. and Kim, W., "Versions and Change Notification in an Object-Oriented Database System," in *ACM/IEEE Design Automation Conference*, pp. 275 - 281, 1988.

[20] Chung, M. J., "A Technology Independent Synthesis Design Environment Using VHDL," Tech. Rep. Internal Report, Michigan State University Department of Computer Science, 1990.

[21] Chung, M. J. and Kim, S., "An Object-Oriented VHDL Environment," in *ACM/IEEE Design Automation Conference*, pp. 431 - 436, 1990.

[22] Chung, M. J. and Kim, S., "Configuration Management for Version Control in an Object-Oriented VHDL Environment," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 258 - 261, 1991.

[23] Chung, M. J. and Kim, S., "A Path-Oriented Algorithm for the Cell Selection Problem," in *Michigan State University, Department of Computer Science*, 1993.

[24] Chung, M. J., Rogers, E. and Won, Y., "VHDL in an Object Oriented VLSI Environment," in *CompCon*, pp. 324 - 327, 1987.

[25] Cirit, M. A., "Transistor Sizing in CMOS circuits," in *ACM/IEEE Design Automation Conference*, pp. 121 - 124, 1987.

[26] Clemm, G. M., "The Odin Specification Language," in *International Workshop on Software Version and Configuration Control*, pp. 145 - 158, 1988.

[27] Cohen, E. S., Soni, D. A., Gluecker, R. Hasling, W. W., Schwanke, R. W., and Wagner, M. E., "Version Management in Gypsy," in *Software Engineering Symposium on Practical Software Development Environments*, pp. 201 - 214, 1988.

[28] Consta, R. M. da, "Integrating VHDL," *High Performance Systems*, pp. 75 - 81, Feb. 1989.

[29] Dai, Z.-j. and Asada, K., "MOSIZ: A Two-Step Transistor Sizing Algorithm based on Optimal Timing Assignment Method for Multi-stage Complex Gates," in *IEEE Custom Integrated Circuits Conference*, pp. 17.3.1 - 17.3.4, 1989.

[30] Daniell, J. and Director, S. W., "An Object-Oriented Approach to CAD Tool Control," in *ACM/IEEE Design Automation Conference*, pp. 197 - 202, 1989.

[31] Detjens, E., Gannot, G., Rudell, R., and Sangiovanni-Vincentelli A., and Wang, A., "Technology Mapping in MIS," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 116 - 119, 1987.

[32] Enomoto, K., Nakamura, S., Ogihara T., and Murai, S., "LORES-2: A Logic Reorganization System," *IEEE Design and Test of Computers*, vol. 2, pp. 35 - 42, Oct. 1985.

[33] Estublier, J., "Configuration Managment: The Notion and the Tools," in *International Workshop on Computer-Aided Software Engineering*, pp. 38 - 61, 1988.

[34] Feldman, S. I., "Make–A Program for Maintainingg Computer Programs," *Software-Practice and Experience*, vol. 9, pp. 255 – 265, Apr. 1979.

[35] Fiduk, K. W. Fiduk, Kleinfeldt, S., Kosarchyn, M., and Perez, B. E., "Design Methodology Management - A CAD Framework Initiative Perspective," in *ACM/IEEE Design Automation Conference*, pp. 278 – 283, 1990.

[36] Garey, M.R. and John, D.S., *Computers and Intractability : A guide to the theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[37] Geus, A. J. de and Gregory, D. J., "The Socrates Logic Synthesis and Optimization System," in *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation,* eds. G. De Micheli, A. Sangiovanni-Vincentilli, and P. Antognetti, Martinus Nijhoff Publisher, Boston, MA, pp. 473 – 498, 1986.

[38] Granacki, J., Knapp, D., and Parker, A., "The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface," in *ACM/IEEE Design Automation Conference*, 1985.

[39] Gupta, R., Cheng, W., Gupta, R., Hardonag, I., and Breuer, M., "An Object Oriented VLSI CAD Framework," *IEEE Computer*, vol. 22, pp. 28 – 38, May 1989.

[40] Hamer, P. van den and Treffers, M.A., "A Data Flow Based Architecture for CAD Frameworks," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 482 – 485, 1990.

[41] Horowitz, E. and Sahni, S., *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[42] *IEEE Standard VHDL Language Reference Manual.* New York, NY, 1988.

[43] Jacome, M. F. and Director, S. W., "Design Process Management for CAD Frameworks," in *ACM/IEEE Design Automation Conference*, pp. 500 – 505, 1992.

[44] Janni, A. Di, "A Monitor for Complex CAD systems," in *ACM/IEEE Design Automation Conference*, pp. 145 – 151, 1986.

[45] Kaplan, I. and Miller, M., *Modula-2 programming.* Hayden Book Company, Hasbrouck Heights, NJ, 1986.

[46] Katz, R. H., "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, vol. 22, pp. 376 - 408, Dec. 1990.

[47] Katz, R.H. Bhateja, R., Chang, E.E., Gedye, D., and Trijanto, V., "Design Version Management," *IEEE Design and Test of Computers*, vol. 4, pp. 12 - 22, Feb. 1987.

[48] Katz, R. H., Lehman, T. J., "Database Support for Versions and Alternatives of Large Design Files," *IEEE Transactions on Software Engineering*, vol. 10, pp. 191 - 199, Mar. 1984.

[49] Kim, S. and Chung, M. J., "Configuration Management for CAD/VLSI Design and its Support for Design Process Management," in *International Conference for Advancement of Science and Technology in Korea*, 1993.

[50] Kim, S. and Chung, M. J., "A Constraint-Driven Approach to Configuration Binding in an Object-Oriented VHDL Design Environment," in *International Conference on Computer Hardware Description Languages and their Applications*, pp. 359 - 374, 1991.

[51] Kim, S. and Chung, M. J., "A Constraint-Driven Version Control in an Object-Oriented VHDL CAD Environment," in *IFIP WG 10.2 International Workshop on Electronic Design Automation Frameworks*, 1990.

[52] Klahold, P. S. and Wilkes, W., "A General Model of Version Management in Databases," in *VLDB*, pp. 319 - 327, 1986.

[53] Ku, D. and Micheli, G. D., "High Level Synthesis and Optimization Strategies in Hercules and Hebe," in *European ASIC Conference*, pp. 124 - 129, 1990.

[54] Lampson, B. W. and Schmidt, E. E., "Oranizing Software in a Distributed Environment," *SIGPLAN Notices*, vol. 18, pp. 1 - 13, June 1983.

[55] Leblang, D. B. and Chase Jr., R. P., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices*, vol. 19, pp. 104 - 112, Oct. 1984.

[56] Lehman, M. M., "Process Models, Process Programs, Programming Support," in *International Conference on Software Engineering*, pp. 14 - 16, 1987.

[57] Lin, S. Lin, Marek-Sadowska M., and Kuh, E. S., "Delay and Area Optimization in Standard-Cell Design," in *ACM/IEEE Design Automation Conference*, pp. 349 - 352, 1990.

[58] Li, W. et al., "The Circuit Implementation Problem," in *ACM/IEEE Design Automation Conference*, pp. 478 - 483, 1992.

[59] Lis, J. S. and Gajski, D. D., "Synthesis From VHDL," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 378 - 381, 1988.

[60] Liu, L.-C., Wu, P.-C., and Wu, C.-H., "Design Data Management in a CAD Framework Environment," in *ACM/IEEE Design Automation Conference*, pp. 156 - 161, 1990.

[61] Mahler, A. and Lampen, A., "shape - A Software Configuration Management Tool," in *International Workshop on Software Version and Configuration Control*, pp. 228 - 243, 1988.

[62] Marschner, E., "VHDL Design Environment," *VLSI Systems Design*, pp. 40 - 49, Sept. 1988.

[63] Narayanaswamy, K. and Scacchi, W., "Maintaining Configurations of Evolving Software Systems," in *International Conference on Software Engineering*, pp. 403 - 408, 1988.

[64] Parsave, K. et al., *Intelligent Databases.* Wiley, 1989.

[65] Perry, D. E., "Version Control in the Inscape Environment," in *International Conference on Software Engineering*, pp. 142 - 149, 1987.

[66] Rochkind, M. J., "The Source Code Control System," *IEEE Transactions on Software Engineering*, vol. 1, pp. 364 - 370, Dec. 1975.

[67] Saunders, L. F., "The IBM VHDL Design System," in *ACM/IEEE Design Automation Conference*, pp. 484 - 490, 1987.

[68] Schwanke, R. W. et al., "Configuration Management in BiiN SMS," in *International Conference on Software Engineering*, pp. 383 - 393, 1989.

[69] Shihsha, T., Kubo, T., Hikosaka, M., Akiyama A., and Ishihara, K., "POLARIS: Polarity Propagation Algorithm for Combinational Logic Synthesis," in *ACM/IEEE Design Automation Conference*, 1984.

[70] Shyu, J., Sangiovanni-Vincentelli, A., Fishburn, J,, and Dunlop, A., "Optimization-Based Transistor Sizings," *IEEE Journal of Solid State Circuits*, vol. 23, pp. 400 - 409, Apr. 1988.

[71] Siepmann, E. and Zimmermann, G., "An Object-Oriented Data Model for the VLSI Design System PLAYOUT," in *ACM/IEEE Design Automation Conference*, pp. 814 - 817, 1989.

[72] Thomas, D. E. et al., "The System Architect's Workbench," in *ACM/IEEE Design Automation Conference*, pp. 337 - 343, 1988.

[73] Tichy, W., "Design, Implementation and Evaluation of a Revision Control System," in *International Conference on Software Engineering*, pp. 58 - 67, 1982.

[74] Ullman, J. D., *Principles of Database And Knowledge-Based Systems, Vol 1.* Computer Science Press, 1988.

[75] Valdes, J., Tarjarn, R., and Lawler, E., "The Recognition of Series and Parallel Digraphs," *SIAM Journal on Computing*, vol. 11, pp. 298 - 313, May 1982.

[76] VanderZanden, N. and Gajski, D., "MILO: A Microarchitecture and Logic Optimizer," in *ACM/IEEE Design Automation Conference*, pp. 403 - 408, 1988.

[77] Vasudevan, V., Mathys, Y., and Tolar, J., "DAMOCLES: An Observer-Based Approach to Design Tracking," in *ACM/IEEE International Conference on Computer-Aided Design*, pp. 546 - 551, 1992.

[78] Wilkes, W., Klahold, P., Schlageter, G., "Complex and Composite Objects in CAD/CAM Databases," in *International Conference on Data Engineering*, pp. 443 - 450, 1989.

[79] Winkler, J. F. H., "Version Control in Family of Large Programs," in *International Conference on Software Engineering*, pp. 150 - 161, 1987.

[80] Wolf, Wayne H., "How to Build a Hardware Description and Measurement System on an Object-Oriented Programming Language," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 288 - 301, Mar. 1989.

[81] P. van der Wolf and T. van Leuken, "Object-Type Oriented Data Modeling for VLSI Data Management," in *ACM/IEEE Design Automation Conference*, pp. 351 - 356, 1988.

[82] Yang, S., *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0.* Microelectronics Center of North Carolina, 1991.