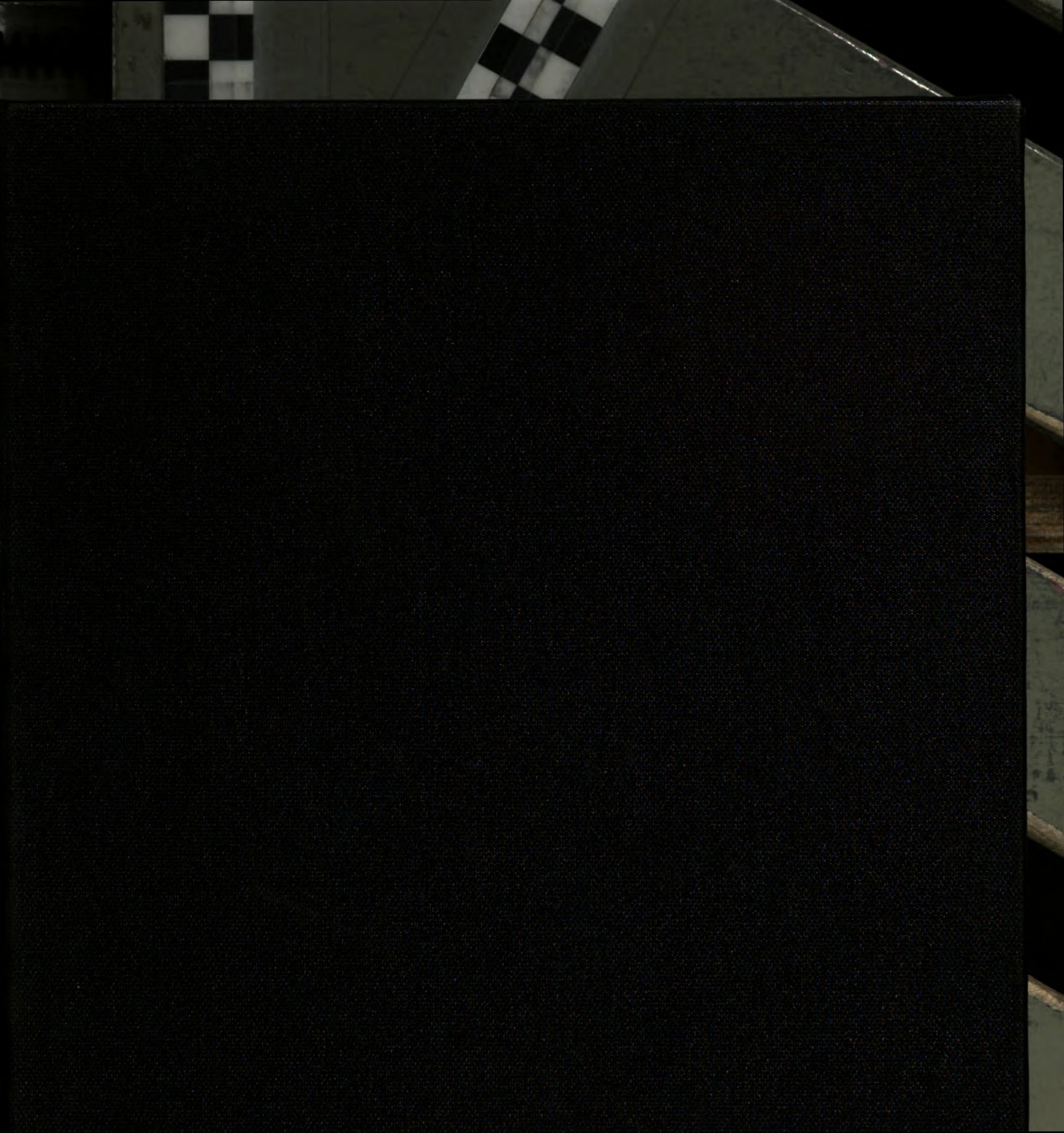


This is to certify that the

dissertation entitled

A DISCIPLINE INDEPENDENT FRAMEWORK FOR ENGINEERING DESIGN

IS

presented by

REID BALDWIN

has been accepted towards fulfillment
of the requirements for

PH.D. degree in PHILOSOPHY

Major professor

Date July 20, 1994

A Discipline Independent Framework for Engineering Design

By

Reid A. Baldwin

A Dissertation

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

Doctor of Philosophy

Department of Computer Science

1994

Moon Jung Chung

ABSTRACT

A Discipline Independent Framework for Engineering Design

By

Reid A. Baldwin

Design frameworks are software environments that integrate the many application tools used by engineers. These frameworks must manage the enormous quantity and variety of data generated. A discipline independent design framework was developed which facilitates the interdisciplinary interaction on which concurrent engineering depends. This framework provides services needed by most engineering disciplines, such as:

- version control,
- configuration management,
- constraint management,
- change notification, and
- methodology management.

Object oriented inheritance is employed to add discipline specific functionality to the discipline independent framework by creating new classes for representation formats such as schematic, solid model, etc.

A unique aspect of this framework is support for products with many user selectable options. This support facilitates the design of versatile, parameterized product families which can be quickly customized to meet customer needs. In addition to increasing the products' appeal to end users, parameterized options allow the predesign of major components before detailed specifications are developed, reducing time to market. The support provided includes facilities to:

- describe what options are offered and describe legal combinations,
- describe how the structure and properties depend upon those options, and
- verify constraints over a large set of combinations.

TABLE OF CONTENTS

LIST OF FIGURES

Part I

Introduction

CHAPTER 1

Introduction

"Hey, Shelley, were the interface specs you gave me for the standard model or the upgrade?"

"You're not still using those specs are you? Those are version 6B. We're up to 8C now."

"But I'm working on a retrofit, I need what was in production in March."

"Oh, I don't think I have those. Try discerning them from Joe's schematic."

"Joe has schematics for 15 different controllers and doesn't remember which one was used."

"Are you doing the retrofit for that problem with the 18 mm aperture? Everybody orders the 21 mm aperture anyway. That's why it wasn't in the test set."

How much time do engineers spend in conversations like this? A lot more than they would like and certainly a lot more than those paying their salaries would like. What can be done about it?

Design frameworks are computing environments that operate at a higher level than the operating system and integrate the application tools used by designers, analysts, and managers. The primary functions of frameworks are:

Design Data Management - managing the enormous amount of data generated during the design process such that designers can easily find the data they need,

Design Methodology Management - managing the design process such that the most appropriate procedure is followed and important steps are not overlooked, and

Services - miscellaneous functions that are enabled by the integration of design activities and data, such as checking constraints, choosing components that satisfy constraints, and notifying designers of changes that influence them.

Design data management includes storing *meta-data* - data about the relationships among data entities [1, 2]. Important relationships such as "represents the same thing as", "is derived from", "is a type of", and "is a component of" are easily lost when designers manage the data without the assistance of a design framework.

Methodology management includes functionality to help designers determine what sequence of tools to use to complete a task. It enables more sophisticated tracking of what is being done, reducing the probability that important steps will be left out.

Many services which were previously performed by stand-alone tools or were not performed at all can be performed better within a design framework. Examples include constraint management and checking, automated component selection, release control, and change notification.

1.1 Characteristics of the design process

Bond and Ricci [3] describe the design process used by a large corporation to design aircraft. Their observations are also applicable to most other product lines. Some of their conclusions are that:

1. Design proceeds by the cooperation of specialists (specialists teams or departments).

2. Each specialist has his own model of the design and may use several different models or partial models for different purposes.

3. Specialists have limited ability to understand each others models. Specialists communicate using a shared vocabulary, but not necessarily shared technical knowledge.

4. Design proceeds by successive refinement of the models which are coordinated and updated together.

5. The design decisions, which are acts of commitment and model refinement, are negotiated by the specialists among themselves.

The design team must specify not only the structure of the design, but the manufacturing process, test procedures, and operating procedures. The resulting design must meet diverse criteria such as performance, cost, aesthetics, and safety. These various aspects of the design are often strongly interdependent, but are usually managed by different individuals or even different departments.

Previously, the design process was largely sequential; marketing determined requirements, the design department specified the products structure, then the manufacturing department decided how to build it and the maintenance department decided how to repair it. Very little information traveled backwards during the process. It was very expensive to modify the design or requirements to incorporate manufacturing or maintenance considerations. *Concurrent engineering* is the practice of developing the manufacturing plans and maintenance plans simultaneously with the design of the product itself. Manufacturability, maintainability, and other "ilities" can

then be incorporated into the product design. If certain requirements are determined to be excessive cost drivers, the requirements can even be modified.

Obviously, communication between designers is crucial in order to perform the necessary trade-offs. Facilitating this communication is an important goal of design frameworks. Existing frameworks are oriented toward specific engineering disciplines, such as VLSI design, structural design, mechanism design, etc. Each framework focuses on integrating only the types of data that arise in that discipline. Since many products require inter-disciplinary design, separate frameworks create an impediment to concurrent engineering.

Complex designs can be synthesized either bottom up or top down. In bottom up synthesis, components are designed first and then integrated to achieve the desired function. In top down synthesis, the overall design is considered first. The components are designed later to meet the constraints imposed by the system design. In practice, both methodologies are combined. Component designers produce designs which do not have specific applications identified, but which are likely to have many uses, such as fasteners, motors, and switches. System designers perform top down design in response to established specifications. The system designer attempts to structure the design such that it uses existing components whenever possible.

The top down design process for mechanical systems is discussed in [4]. In top down design, it is important to be able to represent a part that has not been designed in detail. In mechanical design, representing parts that have not yet been defined may imply representing the three dimensional space that will be occupied by the part [4]. In VLSI design, representing these parts may imply representing their behavior without regard to structure.

There are several different types of design problems. Navathe et. al. classify design problems according to how previous designs are used [5]. In *design from first principles*, no knowledge of previous solutions of similar problems is used. In *design by*

modification, a design for a similar problem is used and modified to meet the slightly different requirements. *Design by selection* implies using the general arrangement of parts from a previous design, but selecting different designs for the components. The designs for the components may exist in a database or may be generated recursively by selection. Most design work is either by modification or by selection. Brown and Chandrasekaran have proposed a similar classification of design problems, based on the degree to which previous design procedures can be used [6]. According to this classification, most design activity is "routine design" in which design procedures that worked in the past for similar products may be used.

Design would be simpler if detailed specifications were always available. Unfortunately, the specific needs of the customer are often not known to the designer. Component designers rarely know the specific needs of their customers, and system designers, such as car designers who must satisfy a wide variety of customers, often face the same problem. To succeed in this environment, it is necessary to design families of products in order to maximize the likelihood of meeting the customers' needs. The product family offers the customers many options, such as size, power, and features. Each different combination of optional features is called a *variant.* Representing the design data for families of products is much more difficult than representing a single product. When applied by component designers, the practice of offering options can dramatically reduce time to market. Offering a choice of parameters allows the component designer to begin long before the system designer sets exact specifications while maintaining a high probability of meeting the eventual specifications.

In [7], Whitney notes that careful design of variants dramatically reduced manufacturing costs for Nippondenso. The designs for a line of panel meters were changed such that the interfaces between the parts were always consistent. The result was a single design with 40 variants that could be built by a single assembly machine.

Instead of requiring 48 different designs for the 6 parts, only 17 were needed with the new design, reducing the inventory control expense.

1.2 Characteristics of design data

The design of a complex product or a line of products involves the creation of an enormous amount of data. Before examining how a framework can assist designers in managing this data, let us discuss some characteristics of the data.

Engineering data can be organized along several different dimensions, as described by Katz [1]. First, all the data which describes the same physical entity should be organized such that it can be treated as a collection. Second, a designer may wish to look at various versions of the same information in order to trace the design history or see what alternatives have already been explored. Third, a classification hierarchy is useful so that similar entities, such as fasteners, can be treated as a collection. Finally, complex designs are frequently decomposed into simpler designs which interact, yielding a component hierarchy.

1.3 Outline

A framework has been developed which addresses many of these problems. This framework differs from others in four primary aspects:

1. The data model is discipline independent and extensible. New representation formats can be integrated into the data model using the object oriented concept of inheritance. A message handling mechanism allows engineers to extract information from the data generated by others without an intimate knowledge of the representation format. This feature facilitates the inter-disciplinary communication which is so crucial to concurrent engineering.

2. Constructs are provided for modeling products with optional content. Formal treatment of optional content provides much more effective management of product families. Without this modeling capability, many of the benefits of frameworks were not available to firms which depend on offering users options in order to be competitive.

3. The framework provides advanced support features which enable an interdisciplinary team of designers to create and verify complex designs. These features include constraint documentation and checking, automated component selection, and change notification. Although these services are provided in other frameworks for products without optional content, new techniques are necessary to address products with options.

4. The framework includes features for managing the design process as well as the design data. These process management features guide designers through an appropriate design process and ensure that important design steps are not inadvertently omitted. These features also help capture the relationships between data entities without relying on designers to enter relationships explicitly.

The main body of this dissertation is organized into three parts, which describe data management, framework services, and process management respectively.

Part II describes the data management features of the framework. Chapter 2 provides an overview of the data model and describes how the component hierarchy, derivation hierarchy, and classification hierarchy are handled. Chapter 3 describes the constructs and algorithms which support the modeling of products with optional content.

Part III describes services provided by the framework. Several of these services require the ability to compute property values for designs. Chapter 4 describes a

flexible mechanism for designers to specify formulas for these property values. Chapter 5 describes how designers may verify a design by expressing constraints which are checked for all variants of a product family. Chapter 6 describes a framework service which assists designers in choosing designs for components such that constraints are satisfied. Unlike the other framework services, which operate on meta data, the automated component selection algorithms described in Chapter 6 directly manipulate the design data. Once designs have been expressed and verified, designers must be informed of certain changes made by other designers in order to keep the design data consistent. Chapter 7 describes how change notification services can be provided within the framework.

Finally, part IV describes framework support for controlling the design process itself. Chapter 8 describes a formal representation of acceptable design processes. An execution environment which uses this representation to help designers choose the most appropriate process is discussed in Chapter 9. When sufficient knowledge is encoded, this execution environment may even perform the more routine portions of the design process automatically.

To assist the reader with the terminology used, a glossary is provided in Appendix A. Appendix B describes the language used to express option offerings, formulas, and constraints. The implementation of persistent objects, which are used to store the design data, is presented as Appendix C.

Throughout this dissertation, a manufacturer of small aircraft engines is used as an example [8]. This company illustrates many of the typical challenges encountered by a modern manufacturing enterprise. This company takes pride in being able to customize its engines to suit the needs of each buyer. The customer can choose from several displacements, ignition systems, carburetors, oil coolers, exhaust systems, etc. and may select many optional features such as hydraulic lifters, an alternator, and a starter.

Part II

Design Data Management

CHAPTER 2

Data Model

A *data model* is a set of structures which organize the data and capture the relationships among data entities. The first section of this chapter provides a survey previous engineering data models. The following section provides an overview of the proposed data model. Finally, Sections 2.3 through 2.5 discuss how the proposed data model captures the component hierarchy, derivation hierarchy, and classification hierarchy, respectively.

2.1 Previous work *

The majority of previous data management frameworks have concentrated on the specific needs of a single engineering discipline such as VLSI design [1, 2, 9, 10], mechanical design [4], building design [11], or chemical process plant design [12]. Sherpa [13] is the only system that currently supports multiple disciplines. Sherpa plays the role of an electronic library for engineering documents, but does not manipulate any of the data within the documents.

*The terminology in the field of engineering data models is not well defined. Different authors frequently use different words for the same concept and the same word for different, but related, concepts. The words *configuration* and *workspace* are particularly troublesome. When describing the work of others, their terminology is used.

Much of the early research in version control was performed in the context of software engineering. Version control systems developed for software engineering only address files of text and do not interpret the contents of the files. These systems concentrate primarily on the efficient storage of many versions by storing only the changes between versions [14, 15]. A check-in / check-out mechanism is employed. New versions are created whenever previously checked out designs are modified and checked back in. All modifications are made outside the system. One such system, RCS [14], keeps track of an acyclic directed graph of versions called a *revision tree.* Since the system keeps track of what was changed between versions, it can often merge versions automatically by combining the changes.

When the data is divided into several related files, version control is not sufficient. A version of a file must be matched up with the correct versions of related files. These groups of versions are called *configurations.* RCS [14] keeps track of configurations by allowing branches in the version history of each file to be named. If an alternative requires changes to several files, the branches for that alternative in each revision history would be given the same name. When the name is supplied as an optional argument of the check-out command, the latest version in that branch is retrieved. Using the same name when checking out several files ensures that the files are compatible.

The Version Server [1], developed by Katz et. al. for managing VLSI design data, uses a check-in / check-out model similar to software configuration management systems. Representation objects must be checked out from a *workspace* for use and new versions are created when representation objects are checked in. In this system, workspaces are logically separate data bases. Three types of workspace are identified and different rules are enforced regarding check-in and check-out. *Archive* workspaces are reserved for versions that are safe for use in any design. Designs must be thoroughly verified before check-ins are allowed. Anyone is allowed to check-out

an objects from an archive workspace. *Private* workspaces are intended for use by a single individual. Only that individual is allowed to check-out objects. However, that individual can check-in objects without any verification. *Group* workspaces allow several designers to share data while they integrate their work.

Katz et. al. clearly identify three types of relationships that exist between objects in a design data base: composition, derivation, and equivalence [1]. The objects which directly contain the design data, called *representation* objects, are identified by name, version number, and type. Several types of *structural* objects are used to record the various relationships. *Composite objects*, which capture the component hierarchy, are formed from representation objects and other composite objects. All of the objects in a component hierarchy are assumed to have the same format. For example, a component hierarchy might contain either layout data or behavioral data but not a mixture of the two. *Equivalence objects* relate representation objects that describe the same design but have a different type. *Version history objects* group representation objects that have a "derived from" relationship. *Configuration objects* associate a composite object with versions of the composite object's components.

In [16], Katz and Chang propose mechanisms for automatically creating new configuration objects when new versions of a component object are checked in to a workspace. This practice can result in an explosion of configuration objects. They also address the issue of which equivalence objects should be added for the new version. Under standard check-in, the new version is assumed to be equivalent to all of the objects that the object it was derived from was equivalent to. In fact, their validation subsystem does not allow a new version to be checked into an archive workspace unless it can verify that it is equivalent. When several of the equivalent objects are modified together a group check-in is required. The new objects are assumed to be equivalent to each other but not necessarily equivalent to the versions from which the other objects were derived.

Kim [17] proposes a data model specifically for VLSI design. He organizes the data in that discipline into four *levels of abstraction*. *CELL* objects specify only the interface but nothing about the implementation. *DESIGN* objects represent entities whose arrangement of components is determined, but the design of the components is not determined (each component is represented by a CELL object). *CONF* objects represent designs in which at least one of the components is bound to a particular implementation (which is represented by a CONF or DESIGN object). *CLASS* objects organize the CELL objects into a classification hierarchy. His data model is based on the VHDL language, which clearly separates interface, implementation, and configuration data. The data entities that are used in other disciplines, such as a finite element model or a set of empirical equations, often do not fit well into these categories.

There are a variety of ways of referring to versioned objects. Most systems support some fashion of *static binding*, which explicitly specifies which version to access. Most systems also support *dynamic bindings* which allow the "appropriate" version to be determined when the object is accessed. Systems differ widely in how the "appropriate" version is determined. Some systems have a movable pointer that specifies the "current" version for each object. Others allow several such pointers and a mechanism for specifying which pointer to use. These pointers may be automatically adjusted when new versions are created or may be explicitly set by users.

DVSS [18] creates a separate pointer for each *derivation path* (linear chain of revisions) which always indicates the most recent version in that path. A reference that does not specify a path accesses the most recent version in a designated path called the *principal path*. In [19], Beech and Mahbod describe a general, though somewhat cumbersome, mechanism for specifying how references to versioned objects should be handled. An object called a *context* contains user defined functions which are applied to any reference to a versioned object in order to select a particular

version. Static and dynamic bindings are special cases that are expressible in their system. Other possibilities would select versions based on user defined properties.

In [20], Dittrich and Lorie present a powerful mechanism for version selection. They define concepts called *version clusters* which contain a group of versions and ***environments*** which map version clusters to particular versions. One environment is selected by the user at a time. Dynamic bindings reference version clusters and the current environment determines which version to use. Dittrich and Lorie also present a very powerful mechanism for defining these environments, but the definition must be processed before use.

Banks et. al. [21] describe a mechanism, called *workspaces*, for mapping versioned objects to versions that is similar to the environments of Dittrich and Lorie. However, users manipulate the mappings more directly. To facilitate the definition of workspaces that are very similar to existing workspaces, the workspaces are organized into a hierarchy. Each workspace only defines the mappings that differ from its parent workspace. Banks et. al. also utilize a special type of workspace, called a ***checkpoint***, to freeze design states. The mappings in checkpoints cannot be changed and the referenced versions are frozen.

Silva et. al. [22] define a *workspace* as a collection of versions, perhaps including several versions of the same design. A separate concept, called a ***configuration*** contains a subset of the versions in a particular workspace such that at most one version for each design is in the configuration. Each workspace may contain several configurations, one of which is declared to be the current configuration for the workspace. Workspaces are not permanently organized into a hierarchy, but users specify a search order. A search is necessary because the current configurations of some workspaces may not contain any versions of the desired design.

Biliris [23] takes a slightly different approach. Only one version of a design is considered current at a given time. Each version of an object may have one of four

statuses: *in-progress*, *stable*, *frozen*, or *released*. In-progress versions are the only type that may be modified. All bindings in a stable version are dynamic bindings which reference whichever version is current at the time the binding is evaluated. Although stable versions cannot be edited, their properties may change over time because the bindings reference updated designs of components. When an object is promoted to frozen status, all dynamic bindings are converted to static bindings which reference whatever version is current at the time of promotion. Before promotion, the current version of each current component designs must have either frozen or released status. Frozen versions represent a fixed state of the entire component hierarchy. Released versions are like frozen versions that cannot be deleted. A design cannot be promoted to released status until all of its component designs have released status. By recording the time that the current version is changed for every design, it is possible to re-create the design state of the component hierarchy that a stable version represented at some specific time in the past. Biliris also organizes the data into separate databases which are nodes in a directed acyclic graph following the project decomposition. Each version has a home database. A version is readable from a database if there is a path from that version's home database to the current database.

Several previous systems organize designs according to a classification hierarchy. Cornelio and Navathe [5] present an interesting alternative called a *Semantic Association Graph* (SAG). SAGs are acyclic directed graphs that organize designs according to what properties (called *capabilities*) are defined for them. Each node represents a property. Each design is associated with a node. The set of properties defined for that design is given by all ancestor nodes in the graph. The SAG facilitates finding existing designs with properties similar to those desired.

In summary, there are several existing systems which record some of the relationships required for design data: classification, derivation, component, and equivalence.

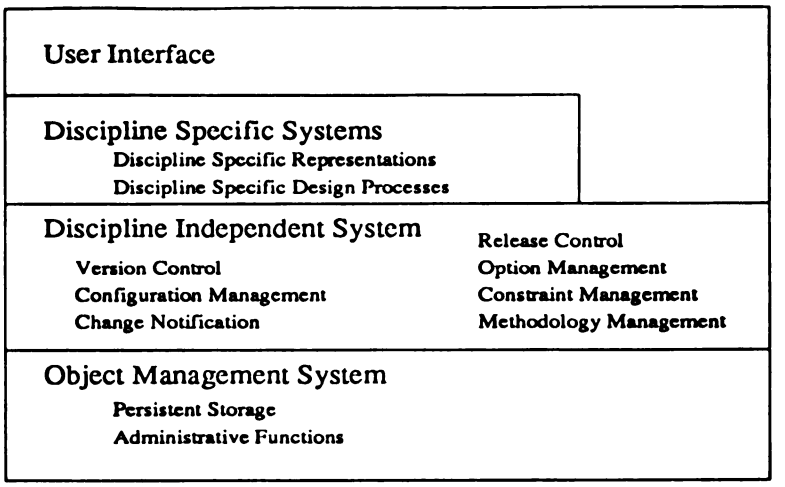


Figure 2.1. Layered Organization

However, none of the systems surveyed records all of these relationships. Furthermore, Sherpa is the only system that is discipline independent.

2.2 Overview

To facilitate the creation of multidisciplinary frameworks, the data model is organized in layers, as shown in Figure 2.1. The discipline independent data model provides services that are required in most or all disciplines, such as option management, version control, and configuration management. Discipline specific functionality, possibly from several disciplines, can easily be added on top of the discipline independent data model. The advantage of the proposed data model is not that the services, taken individually, are better than all other systems, but that the services are all provided seamlessly within a single discipline independent framework.

The data model is illustrated in Figure 2.2. The basic unit of data is a *representation*, which describes a particular aspect of a design, such as its structure, performance, manufacturing process, etc. The various representations that collectively describe the same version of a physical entity are grouped into *design versions*. Design versions have a series of named slots, one for each different type of information

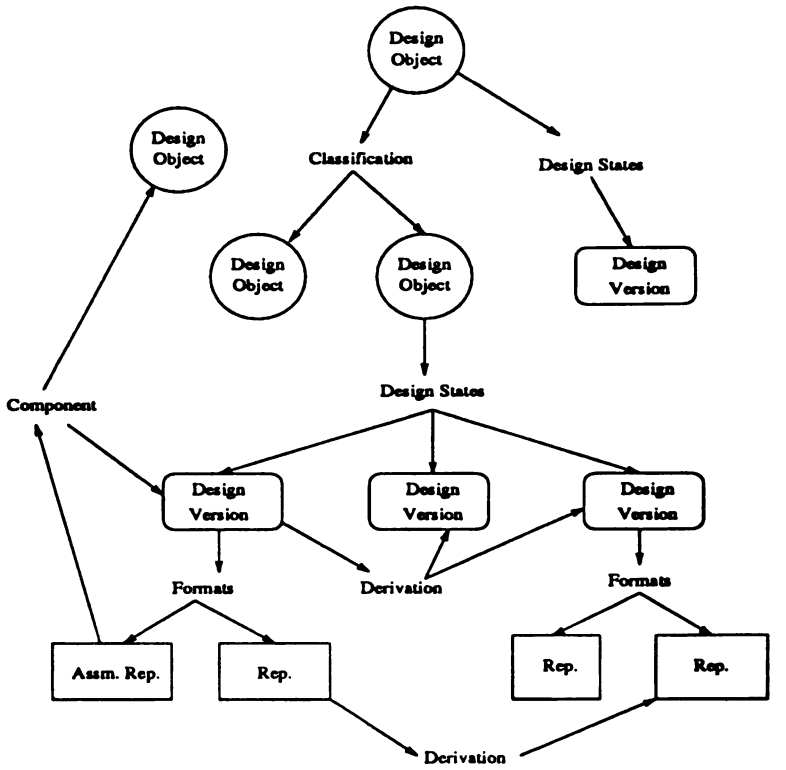


Figure 2.2. Data Model

to be retained about a design. Each slot points to a representation that describes the corresponding aspect. "Derived from" relationships are captured at both the representation and design version levels. Representation and Design Version objects contain flags indicating whether they are currently editable or deleteable. Design Versions also have a flag indicating if it is currently considered valid. The use of this flag is discussed further in Chapter 7.

Unlike most other systems, a check-in / check-out protocol is not used.[†] If data is editable, it can be modified in place, implying that the previous data is lost. New versions are created explicitly when the user decides to save the current state.

The design versions representing the various design states of a particular design are grouped into a *design object.* Design objects are organized into a classification hierarchy. The relationship between design objects and design versions is analogous to

[†]The release operation described in Section 2.4 is similar to the check-in operation.

the relationship between classes and instances in object oriented programming. The non-leaf objects in this classification hierarchy represent generic designs or classes of designs. Like leaf design objects, non-leaf design objects also have a collection of design versions. Of course, the representations in the slots of these design versions contain less specific descriptions of structure and performance than those under leaf designs.

The component relationships are captured by a particular type of representation called an *assembly representation.* Each of these types of objects and relationships is described in more detail in the sections that follow.

Each different data format is called a *representation type.* Discipline specific functionality is added to this framework by creating new representation types using object oriented inheritance. The `Representation` class is used as a base class for defining other representation types. For example, a derived class called `File_Rep` stores design data in a standard computer file (other representation types might store the data in databases). New representation types for solid models, CAD/CAM models, etc. may easily be created as derived classes of the `File_Rep` class, without having to re-program the routines to manipulate files. New methods would be defined to manipulate the design data within the file, probably by invoking the appropriate application program. Figure 2.3 shows some example representation types that could be defined.

Users must be able to utilize the data contained in representations and design versions without detailed knowledge of representation formats. A message handling system has been implemented to support this interaction. Data from different disciplines can interact by exchanging these messages. The semantics of the messages are defined by the implementors of the representation types. For example, the representation type `Solid_Model` would handle a request to compute the volume, etc. as illustrated in Figure 2.4. Specifically, each representation type has a method to

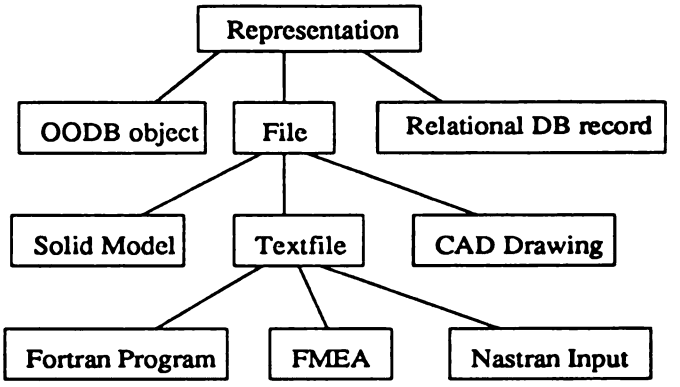


Figure 2.3. Possible subclasses of `Representation`

respond to a message encoded as a text string.[‡] This method decodes the string according to whatever calling syntax the representation programmer determines and takes the appropriate action. If the message cannot be decoded and handled, then the representation type reports failure to the requester. To extract information from a representation, a user needs to consult a message dictionary for the representation type to determine what messages have been implemented. Consulting a message dictionary is easier than learning to interpret the representation semantics.

In a similar fashion, messages may be sent directly to design versions. The design version forwards the message to the representations in its slots. Since each slot generally contains a different type of information, usually only one is able to decode and respond to the message. If none can respond, or if two or more can respond and disagree, the design version reports failure to the requester. The sender of the message must have some knowledge about what representation types are likely to fill the slots of the design version. Designers usually know what type of information to expect in design versions with which they interact.

[‡]Readers may wonder why this could not be accomplished by defining a method for each message type. The problem with this approach is that the user interface cannot provide access to these new methods unless the methods are known to the developer of the interface. The interface would require reprogramming whenever new representation types are added or additional functionality is added to existing types.

```
int
  solid_model::handle_message(char *message, void *answer,
                              environment *home) {
  /* handle messages sent to solid model representations */
  if (!strncmp(message, "get volume", 10)) {
    /* if this is a request to compute volume */
    float *result =  (float *) answer;
    /* response will be a floating point number */
    *result = compute_volume();
    /* do whatever is needed to compute volume,
       probably invoking a tool */
    return 1; /* indicate success to caller */
    }
  ...
  /* similar code for other message types,
     such as "get surface area" */
  return file_rep::handle_message(message, answer, home);
  /* msg doesn't apply to solid models,
     see if parent type can handle it */
  };
```

Figure 2.4. Sample C++ code for message handling

A representation type called `frame` provides a convenient way to access the message handling system for computing properties of a design. Frames are discussed in detail in Chapter 4.

2.3 Component hierarchy

A representation type called an `Assembly` represents designs that are decomposed into simpler designs. An assembly simply contains a list of components and a list of connections, which indicate what the parts are and how they are connected. Specifically, assembly representations contain instances of two classes, `Component` and `Connection`. The `Component` class contains a component name and a binding indicating what design should be used for that component. A `Connection` simply contains a list of components that are to be connected. As illustrated in Figure 2.5, a design version for Cylinder_Head_Asm would contain an assembly representation indicating that it is composed of a cylinder, a cylinder head, spark plugs, and a rocker arm assembly. A design version for Rocker_Asm would contain another assembly representation indicating that rocker arm assemblies contain rocker arms and valve springs.

A reference to another design from within a representation is called a *binding*. Bindings are evaluated to determine what design version object should be referenced. Several different types of bindings are supported. The simplest type of binding, called a *static binding*, allows the designer to specify the design version directly. A more sophisticated type of binding, called a *dynamic binding*, allows the designer to specify only the design object. When dynamic bindings are used, the system determines, at the time the binding is evaluated, which design version would be appropriate. The mechanism for making this determination is discussed in the next section. Users may set a flag indicating that binding evaluations that would reference a design version

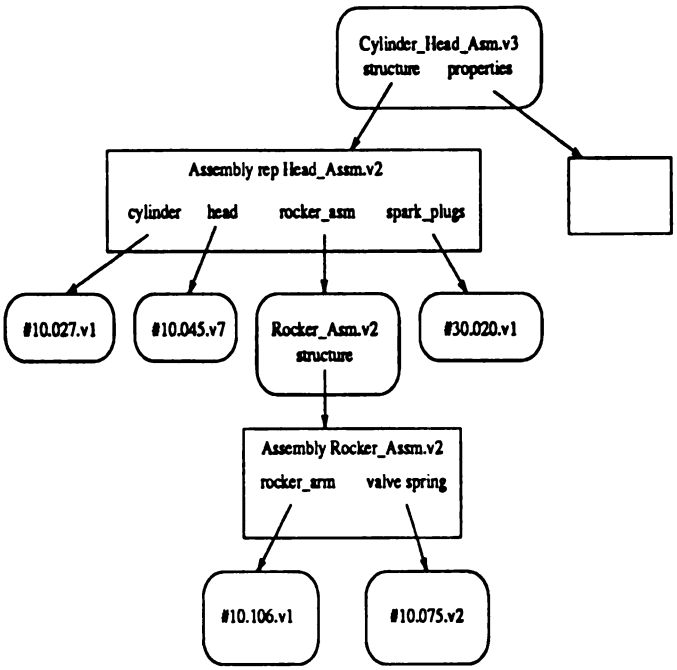


Figure 2.5. Component Hierarchy of Cylinder Head Assembly

with invalid status should fail. This flag prevents the unintentional use of invalid data.

Because each discipline needs to store different information about the structure of a design, the `Assembly` representation type is extensible. For example, electrical engineers may need to indicate which pins on an integrated circuit to connect, and mechanical engineers may need to specify the torque to which a nut should be tightened. Property lists associated with each component and connection provide designers a way to express discipline specific information without the need to do any programming. These properties are simply expressed as < property name, value > pairs. Numeric, string valued, and Boolean properties are all allowed. If property lists are not sufficient to express the discipline specific information, a programmer can extend `Assembly` representations by creating new classes which are derived from the `Component` and `Connection` classes. For example, a representation suitable for

discrete event simulations might use a subclass of `Component` called **`DES_component`** that defines an additional method `handle_event()`.

2.4 Derivation hierarchy

Consider a scenario of an engineer designing a new cylinder for the aircraft engine. As she worked, she created several versions of the design. The geometry of the first version was drawn using a drafting package. A program for a numerically controlled (NC) machine was generated to manufacture the part. She was not pleased with the NC program so she modified the program without changing the part's geometry. Unable to generate a satisfactory NC program for the original geometry, she eventually decided to modify the geometry and generate a new NC program. When she finished, she had five data files representing three different design states. Unless the relationships between files are carefully documented, there is a danger of using old versions or using an NC program that is inconsistent with the geometry.

Under the proposed data model, this design data would be captured by the design version objects and representation objects illustrated in Figure 2.6. The design data is contained directly in representation objects. Each design version object has a series of named *slots* and a pointer to a representation object corresponding to each slot. Collectively, these representation objects describe the design state of the design, but not the states of other designs that are referenced within the representations, such as components. A broader concept of design state is discussed below. A design state is modified by changing which representation is associated with a slot or by modifying the data within one of the representations. Slots can also be added or removed. Version relationships are maintained separately for design versions and representations, although the derivation hierarchies are usually similar.

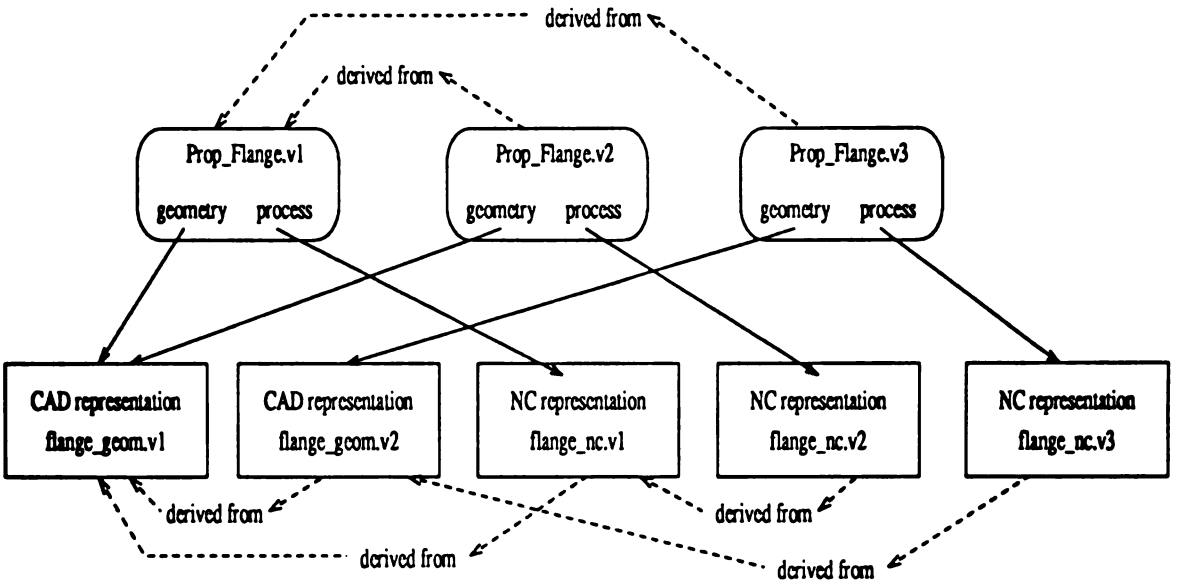


Figure 2.6. Cylinder Derivation Hierarchy

A set of rules, which determine when representations and design versions may be modified or deleted, must be enforced. These rules are:

1. A design version should not be modified when one or more design versions have been derived from it. If the design state described by the design version were changed, the "derived from" relationship would no longer hold. Instead, the designer should create a derived design version and modify the new design version.

2. A representation with derived versions should not be modified for the same reason as above.

3. A representation which is currently referenced by a slot of a design version should not be deleted.

4. If a design version cannot be modified, then no representation which is referenced by a slot of that design version should be modified.

Suppose that the engineer responsible for designing the cylinder head assembly wants to do an analysis of its performance. He must reference the design of the cylinder. Which version of the cylinder design should he use? He would want to use a recent version, yet one that is fairly stable.

Selecting the correct version requires input from both the designer of the part in question and the person performing the analysis. The choice of versions depends upon the intent of the analysis, which is known to the analyst. The objective might be to determine the performance of a proposed design or might be to investigate a failure of a prototype or unit in the field. Input is also required from the designer of the part in question because he would know which version is appropriate once the purpose is defined.

Workspaces provide a convenient mechanism for version selection. My data model adapts the workspace concept described by Banks et. al. in [21]. This mechanism is not as expressive as the mechanisms of Beech and Mahbod [19] or Dittrich and Lorie [20], but is conceptually simpler. Typical design activities do not require the additional expressiveness of the more complex mechanisms.

A workspace is a mapping (possibly a partial mapping) from design objects to design versions. At any point in time, a workspace defines the design state for the complete component hierarchy, called a *configuration*. A separate workspace is defined for each different analysis situation. The designer of each part decides which version is appropriate for each workspace. Specifying that a workspace should map a design object into a particular design version is called *releasing* the design into the workspace. Workspaces are organized hierarchically to facilitate specifying the mappings. The designer of a part only needs to specify the version for a workspace if it differs from the parent workspace. This organization allows the creation of many workspaces without interfering with the work of designers. The analyst specifies a workspace before constructing the configuration. The mapping of the selected workspace is

used to determine which design version to use for the referenced design object. If no mapping is specified for the current workspace, the parent workspace is searched. The process continues until a mapping is found or a workspace does not have a parent. If the entire ancestry of a workspace is searched unsuccessfully, the evaluation fails.

Figure 2.7 shows and example workspace hierarchy and how each workspace maps design objects to design versions. In this example, a workspace called "Project" has been created that contains stable versions of both the cylinder design (version 1) and cylinder head assembly design (version 3). To perform analysis on a new cylinder head assembly (version 4) using the stable cylinder design, the head assembly designer can create a new workspace called "Head_Assm_Test" as a child of "Project" and release the experimental version of the head assembly into that workspace. The analysis can then be performed without disturbing the cylinder designer. However, when an updated cylinder design (version 2) becomes sufficiently stable, it will be released into the "Project" workspace in place of the old version. The new version will then automatically be included in any analysis done by the head assembly designer without any explicit action on his part.

Sometimes a designer needs to freeze the state of referenced designs so that future analysis is possible using exactly the same versions. A special type of workspace, called a *checkpoint*, is used to freeze the state of the complete component hierarchy. Like a regular workspace, a checkpoint maps design objects to design versions. However, all of the design versions used must be frozen and the mapping cannot be modified, which implies that all ancestor workspaces must be checkpoints. Otherwise, a checkpoint could inherit a mapping to a design version which is not frozen. Using the framework, users can quickly create checkpoints as a copy of the current state of a workspace. In Figure 2.7, a checkpoint of the "Project" workspace was made to record the status at the end of 1992, making it easy to determine what has changed since then.

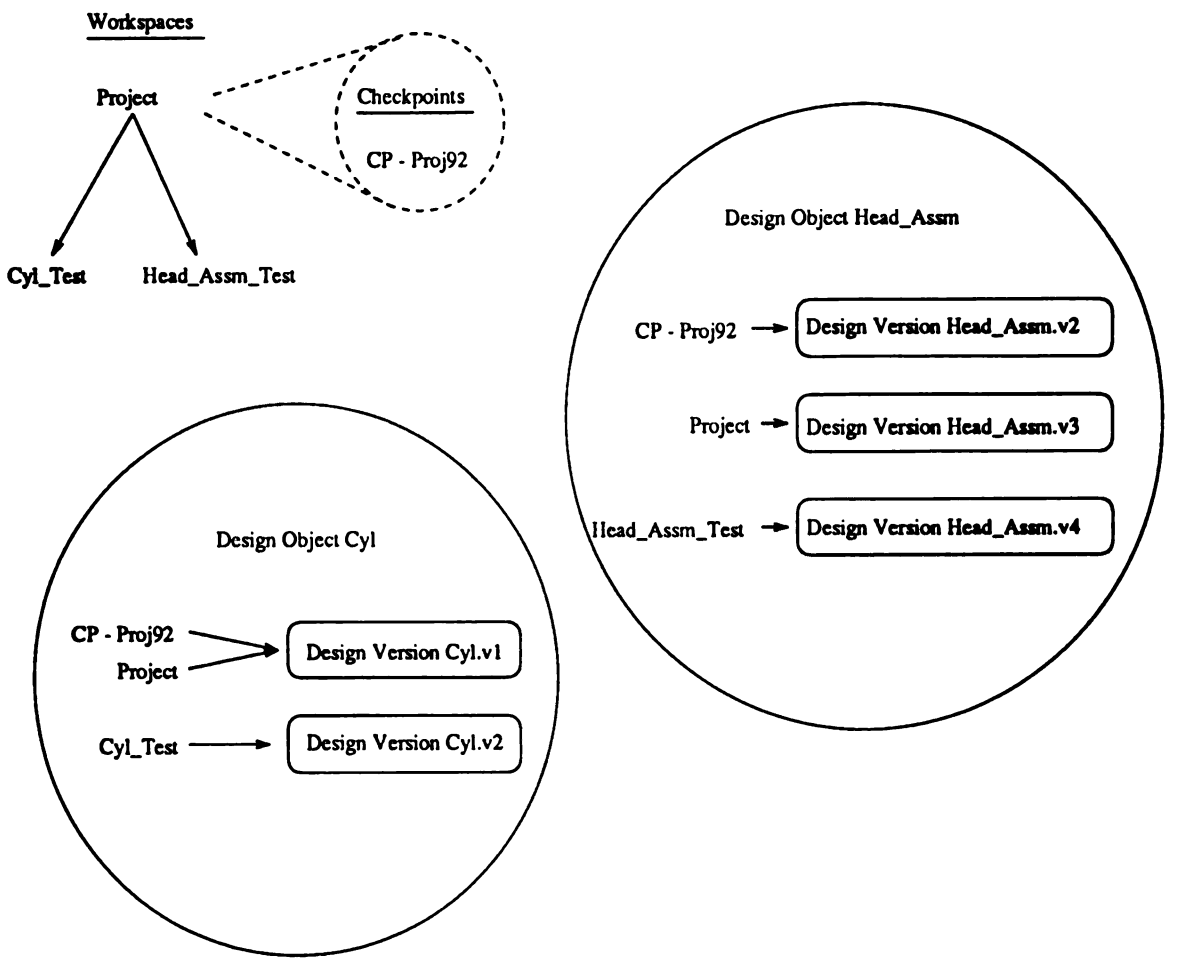


Figure 2.7. Workspaces Map Design Objects to Design Versions

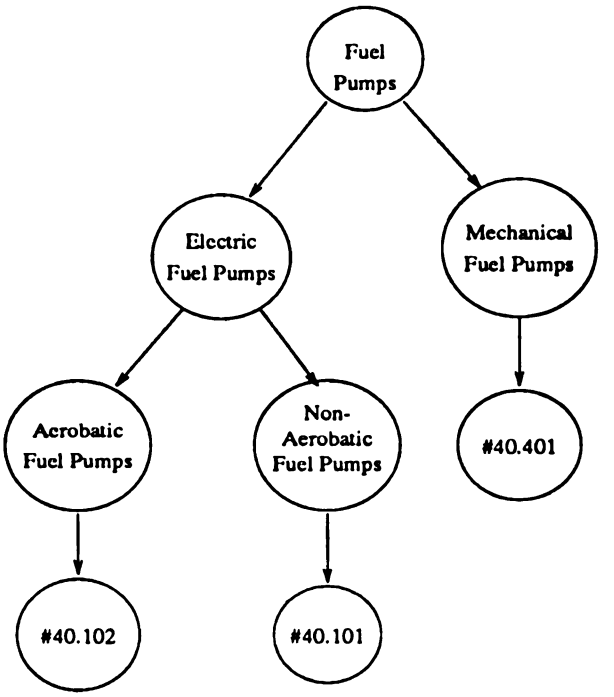


Figure 2.8. Classification Hierarchy of Fuel Pump Design Objects

For storage efficiency, the framework makes the new checkpoint a child of the most recent checkpoint of the copied workspace. Explicit mappings are only needed for design versions that have been added to the workspace since the last checkpoint. If the copied workspace has never been checkpointed, a recent checkpoint of an ancestor workspace may be used.

2.5 Classification hierarchy

Suppose that an engineer is designing a new fuel system for use with the aircraft engines. In the early stages of design, he does not want to decide exactly what fuel pump to specify. Instead, he would like to perform some preliminary analysis assuming a generic model of a fuel pump. When he is ready to specify a particular fuel pump design, his job will be much easier if the available designs are organized into a classification hierarchy, as shown in Figure 2.8.

Design objects are organized into such a hierarchy. Each design object contains a list of other design objects which describe entities that are specializations or generalizations. Each design object has an "is a type of" relationship with its parent design objects. The classification hierarchy is useful in two ways:

1. It organizes the designs for easy browsing, making it easier for a designer to find out about all available designs when selecting a component or creating a similar design.

2. Non-leaf design objects provide a convenient repository for models that apply to many different designs.

New design objects can be created by design engineers without doing any programming.

A list of constraints associated with each design object guides designers looking for designs with certain capabilities. The language for expressing these constraints is described in Appendix B. Constraints can either restrict the situations in which a design may be used or guarantee a certain level of performance. The constraints are automatically inherited by all of the ancestor design objects in the classification hierarchy. The design object `Fuel Pumps` in Figure 2.8 may have the constraints:

- $pressure\_out > pressure\_in$
- $flow\_out = flow\_in$

The design object `#40.102` would inherit these constraints and could add:

- $pressure\_out - pressure\_in > 6$

The added constraints may supersede inherited constraints, but should not contradict them. Constraint checking functionality, presented in Chapter 5, helps designers detect contradictory constraints.

CHAPTER 3

Optional Content

When a family of products with many variants is designed, additional constructs must be added to the data model. Very little work has been done representing families of products that have many variants based on user selectable options, although the practice of designing such families is common [27]. Traditionally, variants have been described by notes on drawings. The drawing illustrates a certain variant, and the notes describe how to modify the design to produce other variants. These textual notes are not amenable to automated processing. It is also very difficult to determine if all of the conceivable variants are correct or even unambiguously specified.

Inventory control experts created representation formats called ***hierarchical bills of material*** and *product structure charts* which provide a formal means of expressing what components should be used for each combination of option choices [28]. These concepts add some formalism and allow automated processing. However, they only represent the list of parts, not the fabrication, assembly, operating, and maintenance information.

The most significant effort to date in the field of representing products with options is the generic bill-of-material product model of Hegge and Wortmann [29]. This extension of bills of material, which are common in industry, allows ***generic products*** to be defined which have a set of variants determined by values for a set of ***parameters.***

Each parameter, like color, has a finite set of possible values. Generic bill-of-material product models, however, are not sufficiently expressive to describe many products. The mechanism for restricting what combinations of parameter values are allowed is also very weak.*

In this chapter, the constructs in the proposed data model that handle products with optional content are described. Section 3.1 describes constructs for specifying what variants are to be offered. Section 3.2 describes constructs for specifying which variants are to be produced or analyzed. Section 3.3 introduces constructs for describing the structure and performance of designs as a function of what options are ordered.

Several of the operations that manipulate these constructs are NP-complete. However, the problem instances that typically occur display characteristics that allow them to be solved practically if sophisticated algorithms are employed. In Section 3.4, a data structure that enables these operations to be performed efficiently in most cases is described along with a description of problem characteristics which influence the efficiency.

3.1 Optional features

For each design object, designers must specify what optional features the customer is allowed to select and what choices are available for each. In the proposed data model, each customer selectable parameter is called an *option variable.* Option variables are automatically inherited by descendant design objects in the classification hierarchy. Each option variable has an enumerated list of possible values. Boolean variables always have the possible values TRUE or FALSE, but designers must specify the list

*A more powerful mechanism for describing *not-allowed* combinations is promised as future work.

of possible values for numeric and string option variables. Option variables are very similar to what Hegge and Wortmann refer to as parameters [29].

Optional features are not necessarily independent. Therefore, the designer may not want to offer all combinations of values for option variables. There are several reasons to restrict the option combinations for a design object:

- Some combinations are not manufacturable or would not perform satisfactorily. For example, ordering a large engine without an oil cooler might not be allowed.
- Some combinations are unlikely to be ordered in sufficient volume to justify the expense of offering them.
- A design object may not offer all of the choices for an inherited option variable. The choices not offered would presumably be available from a sibling design object in the classification hierarchy.

Using a subset of the language described in Appendix B, designers specify the restrictions for each design object as a set of Boolean expressions. Like option variables, option restrictions are automatically inherited by all descendant design objects in the classification hierarchy. The set of legal option combinations for a design object is called the *applicability set.* Generic bill-of-material product models [29] do not offer any powerful constructs for restricting the legal combinations. The only type of restriction is that several components with the same parameters can be forced to use the same value for that parameter. For example, if several components have a parameter `color`, the designer can insist that all of the components be the same color.

Some of the option variables for the aircraft engine would be the numeric variable **Disp** with possible values {2165, 2074, 1915, 1835}, the string variable **$Ign_type** with values {'dual_elec', 'dual_mixed', 'single_elec', 'single_mag'}, and the Boolean variables %Starter, %Alternator, and %Aerobatic. (The first character

of the option variable's name indicates its type.) The applicability set for the line of aircraft engines includes the following restrictions:

`(%Starter -> %Alternator)` indicating that a starter cannot be ordered unless an alternator is also ordered.

`(Disp >= 2074) -> ($Ign_type={'dual_elec', 'dual_mixed'})` indicating that single ignition is not offered with the larger displacements.

The design object for a particular engine design, `red74DX`, would inherit these restrictions and add the following restrictions:

`(%Starter)` indicating that a starter always comes with this particular engine design.

`(Disp = 2074)` indicating that this engine comes in only the displacement 2074 cc.

`($Ign_type = 'dual_elec')` indicating that this engine uses dual electronic ignition.

The added restrictions may make some of the inherited restrictions unnecessary, but they should never be contradictory. If contradictory restrictions are accidentally imposed, it would be detected whenever analysis was performed on the design.

3.2 Selecting option values

Option selection objects specify the option combinations for the top level design in a component hierarchy. Each option selection object specifies a set of option combinations. Option selection objects are associated with particular design objects and are inherited by descendant design objects in the classification hierarchy. To perform an analysis, a designer must specify both an option selection object and a workspace.

An option selection object contains an ordered list of statements. Most of the statements are interpreted as Boolean expressions representing option restrictions.

To determine the set of option combinations from an option selection object, the restrictions are processed in order, progressively narrowing the option combinations. Initially, all combinations in the applicability set are considered† If an option restriction expression is satisfiable by at least one combination under consideration, all combinations that violate the restriction are dropped from consideration. If the restriction is not satisfied by any combination under consideration, it is ignored.

Often, the desired combination of option values differs only slightly from an existing option selection object. To facilitate specifying the options in this situation, some of the statements in an option selection object specify another option selection object (from the same design object or a parent in the classification hierarchy) to include at that point. By specifying the differences from an existing option selection, and then including that option selection at the end, the included option selection is effectively used as a default. When two or more restrictions conflict, the restrictions processed earlier take precedence.

For option selection object A of Figure 3.1, dual mixed ignition would be used, and the bore would be set to 92mm. The first statement of option selection B excludes any option combination which uses dual mixed ignition and a bore other than 94mm from consideration. The second statement indicates that the restrictions of option selection A should be considered next. By the time the second statement of option selection A is reached, it is not satisfiable and is ignored.

To specify a variant using Hegge and Wortmann's generic bill-of-material product model [29], the user must assign values for parameters at all levels of the component hierarchy.

---

†If the object contains contradictory restrictions, the applicability set would be empty, so the contradiction would be discovered at this point.

Option Selection Object A

```
($Ign_type = 'dual_mixed')
(Bore = 92)
```

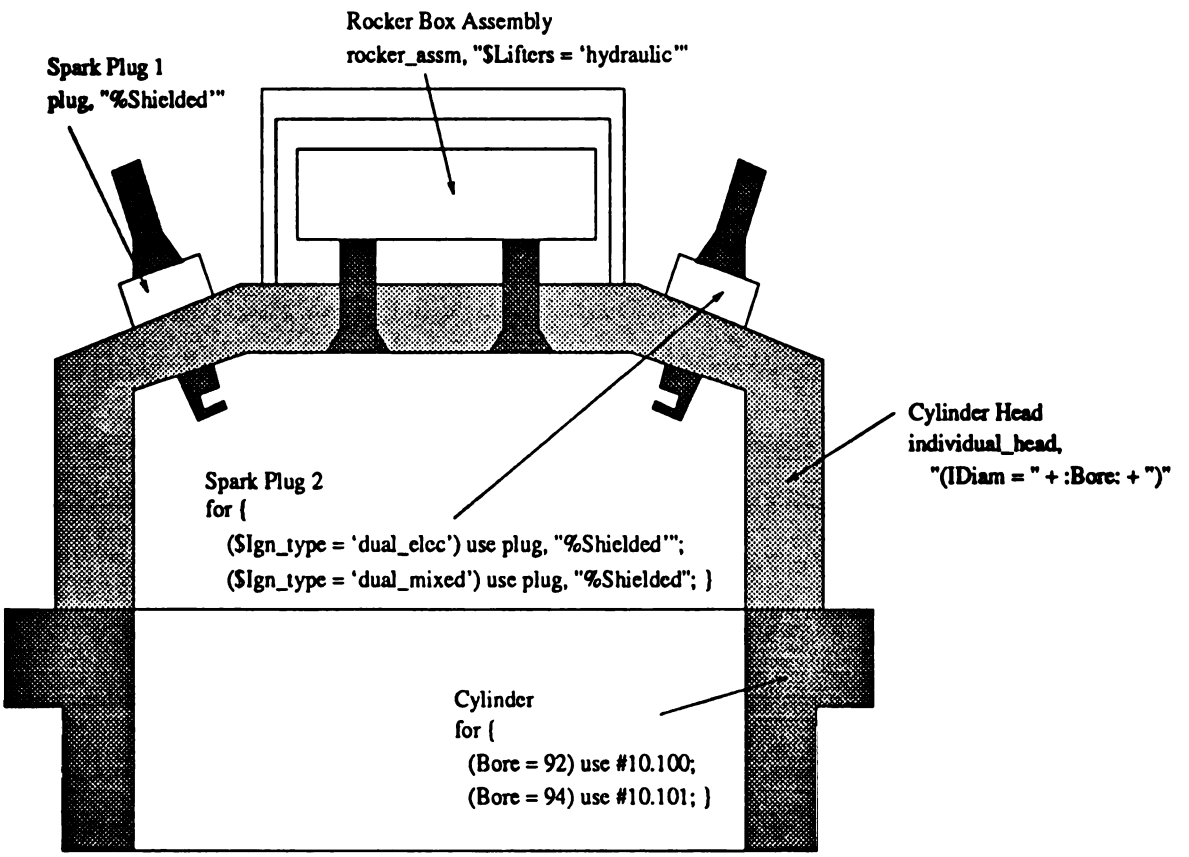
Option Selection Object B

```
($Ign_type = 'dual_mixed') -> (Bore = 94)
#include A
```

Figure 3.1. Example Option Selection Objects

3.3 Specifying designs with options

The option combinations for each component must be specified in the binding as a function of the option combination of the aggregate design. For example, the cylinder head component used for the assembly illustrated in Figure 3.2 should have an inside diameter, `IDiam`, determined by the `Bore` option variable of the referencing design. Options of referenced designs are specified by adding a list of string valued expressions to static and dynamic bindings. These expressions are evaluated using the option combinations for the referencing design. The resulting strings are interpreted as option restrictions which, combined with the restrictions defining the applicability set of the component design, restrict the option values of the component. The binding for cylinder head in Figure 3.2 indicates that when the option variable **Bore** takes the value 92 the design for `individual_head` with the option restriction **(IDiam = 92)** should be used. When more than one option combination is being considered for the referencing design, it may be impossible to evaluate some of the string expressions in which case the binding evaluation fails and the analyst should try the analysis with a smaller set of option combinations.



Applicability Set:

(Bore = {92, 94})

($Ign_type = {'dual_elec', 'dual_mixed'})

Figure 3.2. Bindings of Cylinder Head Assembly

Hegge and Wortmann [29] avoid specifying component options within bindings by requiring the user to specify the parameters of the component directly when specifying the parameters of the aggregate design. A construct called ***phantom generic products*** provides some limited authority for a designer to restrict the option choices of components.

In many cases, the designer may wish to specify a completely different binding depending on what options are selected for the referencing design, such as for the cylinder component in Figure 3.2, which should be a different design depending on `Bore`. *Generalized bindings* allow designers to specify multiple bindings and indicate when each should be used. The designer uses a Boolean expression called a ***case label*** to specify when each binding applies. In this example, the case labels are `(Bore = 92)` and `(Bore = 94)`. Every option combination in the applicability set should satisfy exactly one of the case labels. A generalized binding is evaluated by determining which case applies to the current option combinations and then evaluating the corresponding binding. When more than one option combination is being considered and no case label satisfies them all, the generalized binding cannot be evaluated. Hegge and Wortmann do not provide any constructs similar to generalized bindings [29].

Although bindings are most common in assembly representations, they also appear in other representations. For example, manufacturing instructions may contain a reference to the design of the tooling used. If different tooling is used depending on which options are selected, a generalized binding would be used. The tools or machinery may have optional features or settings, such as different speeds for a drill press. These features can be modeled as options and specified within the bindings.

Another construct for specifying designs with options, called ***conditional inclusion***, is used to include some portion of a representation for some option combinations and not for others. For example, the engine case is cast with a cylinder opening sized for the smallest cylinder bore. These openings must be enlarged when the larger

cylinders are used. The representation describing the manufacturing procedure for the engine case would include the following step:

```
If (Bore = 94), machine cylinder openings to 47mm radius.
```

3.4 Internal representation

In this section, some implementation issues of option restrictions, option selection objects, and generalized bindings are discussed. Although the proposed language is convenient for designers, it must be translated into a data structure that the computer can manipulate efficiently. Users of the framework would not be aware of these details.

The data structure used is an extension of Ordered Binary Decision Diagrams (OBDDs) [30] called *Multiway Decision Diagrams* (MDDs). Standard OBDDs represent Boolean expressions of Boolean variables. MDDs extend them to represent Boolean, string, or numeric valued expressions of enumerated Boolean, string, and numeric variables. Most of the algorithms which construct and operate on MDDs are natural extensions of the corresponding algorithms for standard OBDDs. MDDs are created during the evaluation of option selection objects and generalized bindings, as well as other operations that are discussed in Chapters 5 and 6. MDDs are transient data structures that are deleted after the operation is completed.

Several other extensions of OBDD's have been proposed previously. Algebraic Decision Diagrams (ADDs) [31] extend OBDDs to represent functions with any enumerated domain. However, the functions must still use only Boolean variables. MDDs are an extension of ADDs because the variables may be any enumerated type. Another extension of OBDDs, called Edge Valued Binary Decision Diagrams (EVBDDs) [32, 33] represent integer valued functions of Boolean variables.

An example of a MDD is shown in Figure 3.3. An MDD is a directed acyclic graph. There are two types of nodes: terminal nodes and non-terminal nodes. ***Terminal nodes***

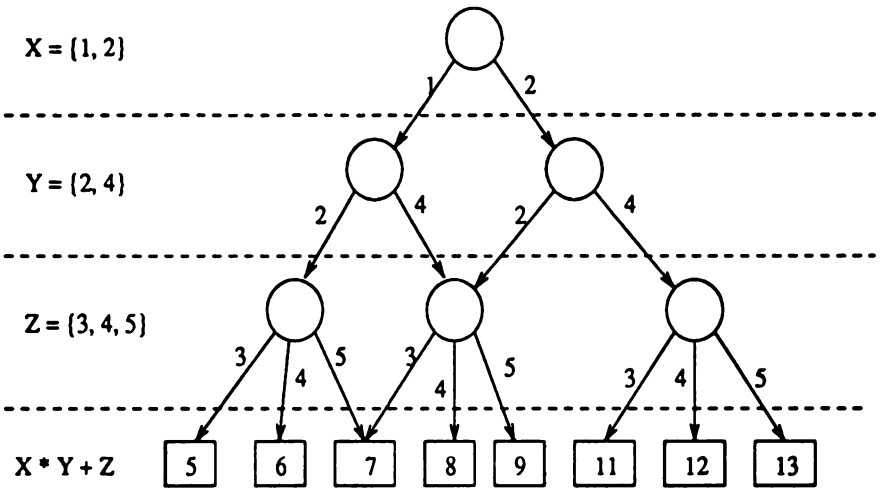


Figure 3.3. An MDD for the expression X*Y+Z

(drawn as rectangles) have values corresponding to possible results of the expression. Each *non-terminal node* (drawn as a circle) is associated with a variable and has an outgoing edge for each possible value of that variable. The arcs leaving a non-terminal node always go to a terminal node or a non-terminal corresponding to a variable later in the ordering. One of the nodes is designated the start node. The value of the expression for particular values of the variables is determined as follows:

1. Begin at the start node.

2. While at a non-terminal node, follow the outgoing edge corresponding to the current value of the node's variable.

3. When a terminal node is reached, the node's value is the value of the expression.

A standard OBDD is a special case of an MDD in which the terminal nodes and all of the variables are Boolean valued.

If there are N variables and variable i has v_i possible values, the worst case has $\prod v_i$ terminal nodes and $O(\prod v_i)$ non-terminal nodes. However, many expressions, such as those that involve a small number of variables and have a small range of results, can be represented by a small number of nodes. The number of non-terminal nodes

required sometimes depends heavily on the ordering of the variables. Determining the optimal ordering is itself an NP-complete problem [30]. An effective hueristic for reducing the number of non-terminal nodes is to keep variables that appear in the same restrictions close together.

Figures 3.4 and 3.5 illustrate the efficiency of this representation. The MDD for the two restrictions (`%Starter -> %Alternator`) and (`Disp >= 2074) -> ($Ign_type={'dual_elec', 'dual_mixed'}`) (Figure 3.4) only requires four non-terminal nodes to represent a set of 72 out of 128 possible combinations of option values. The MDD in Figure 3.5 requires only three nodes to represent the three restrictions (`%Starter`), (`Disp = 2074`), and (`$Ign_type = 'dual_elec'`). This efficiency can be attributed to the fact that each of the restrictions involves only one option variable. In general, MDDs represent sets of restrictions efficiently whenever each restriction either

- involves only one option variable or
- involves a small number of option variables that are close together in the ordering.

The number of non-terminal nodes is likely to grow exponentially in the number of restrictions that violate these conditions. However, the applicability sets that arise in practice rarely contain more than a few restrictions violating these conditions.

The algorithms for manipulating MDDs are straight forward extensions of the corresponding algorithms for OBDDs [30]. The following recursive algorithm computes a new node by combining nodes from two different MDDs using an operator. Given MDDs for two subexpressions, the MDD for the combined expression is computed by combining the start nodes with this algorithm. The variable ordering must be the same for the MDDs being combined. This algorithm is very similar to algorithm Apply in [30], so no proof of correctness is given here.

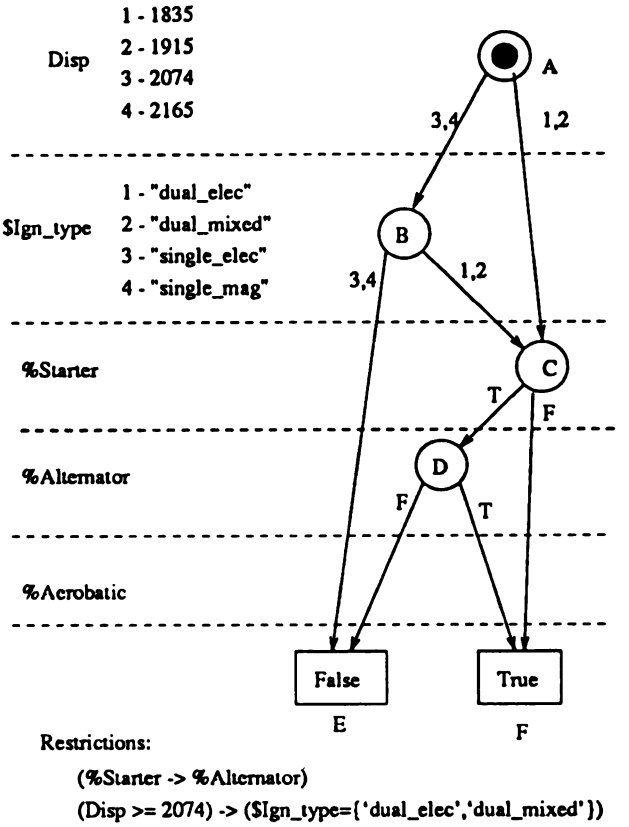


Figure 3.4. MDD for applicability set of engine

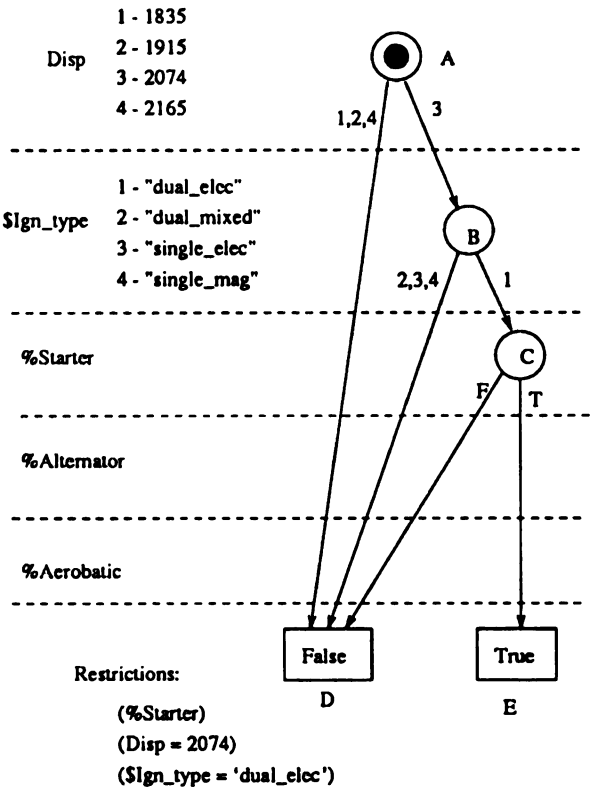


Figure 3.5. MDD for additional restrictions of red74DX

```
Algorithm node_combine

Input:
  One node from each of two MDDs being combined
  The operator being used to combine the MDDs

Output:
  A node from the combined MDD (If the input nodes are
  the respective start nodes, the output node will be
  the start node of the combined MDD.)

1. If both nodes are terminal, the result is a terminal node
   with value determined by applying the operator to the
   values of the two nodes.

2. If they are both non-terminals at the same level, the
   result is a non-terminal at that level with children
   determined by combining their corresponding children.

3. If node1 is a non-terminal at a higher level than node2, the
   result is a non-terminal at the level of node1, with children
   determined by combining node1's children with node2.

4. Similarly, if node2 is a non-terminal at a higher level
   than node1, the result is a non-terminal at the level of
   node2, with children determined by combining node2's
   children with node1.
```

This algorithm is not very efficient because the combine procedure can be called many times on the same pair of nodes. In fact, it is exponential because combining two nodes at level i may require up to v_i recursive calls to combine nodes at level $i-1$. Fortunately, several easy optimizations can be made [30]. The most significant is to keep track of the results of previous calls to combine to avoid repeating the computation on the same pair of nodes. With this optimization the computation time and the size of the resulting MDD is bounded by the product of the sizes of the input MDDs. Figure 3.6 shows the effect of applying this algorithm to the MDDs of Figures 3.4 and 3.5 with the operator AND. The node labeled CD in Figure 3.6 is

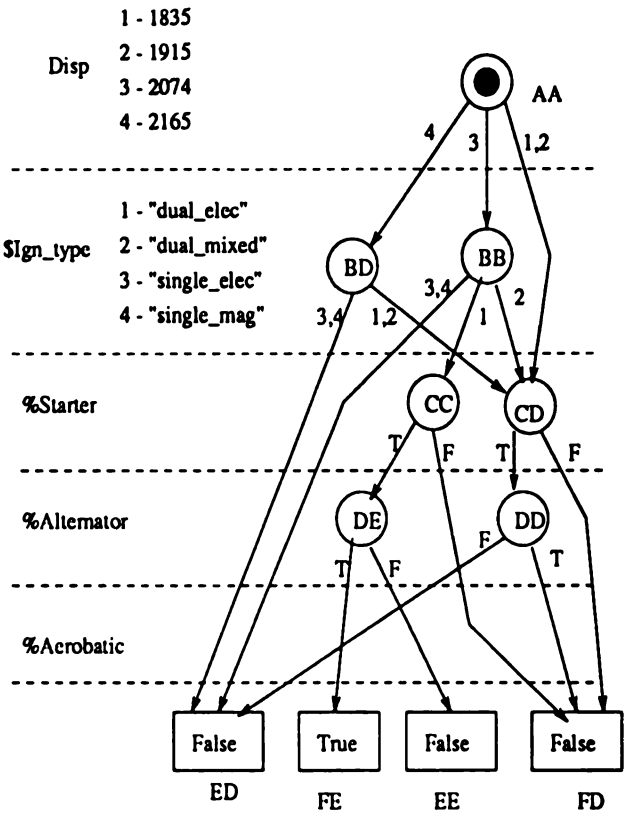


Figure 3.6. MDD produced by combining previous MDDs with AND

the result of combining the node labeled C in Figure 3.4 with the node labeled D in Figure 3.5.

The MDD obtained often has more nodes than necessary. Two terminal nodes are equivalent if they have the same value, such as nodes ED, FD, and EE in Figure 3.6. Two non-terminal nodes, A and B, at the same level are equivalent if for every value, i, of the variable, $A- > child[i]$ is equivalent to $B- > child[i]$. Also, if all of the children of a non-terminal node are equivalent, then the node is equivalent to its children, as is the case for nodes DD, CD, and BD in Figure 3.6. MDDs may be reduced by working from the bottom up and combining equivalent nodes. The result of reducing the MDD in Figure 3.6 is shown in Figure 3.7.

In Chapter 6, an algorithm is presented which needs to build a string expression from an MDD. A string S can be constructed from a non-terminal MDD

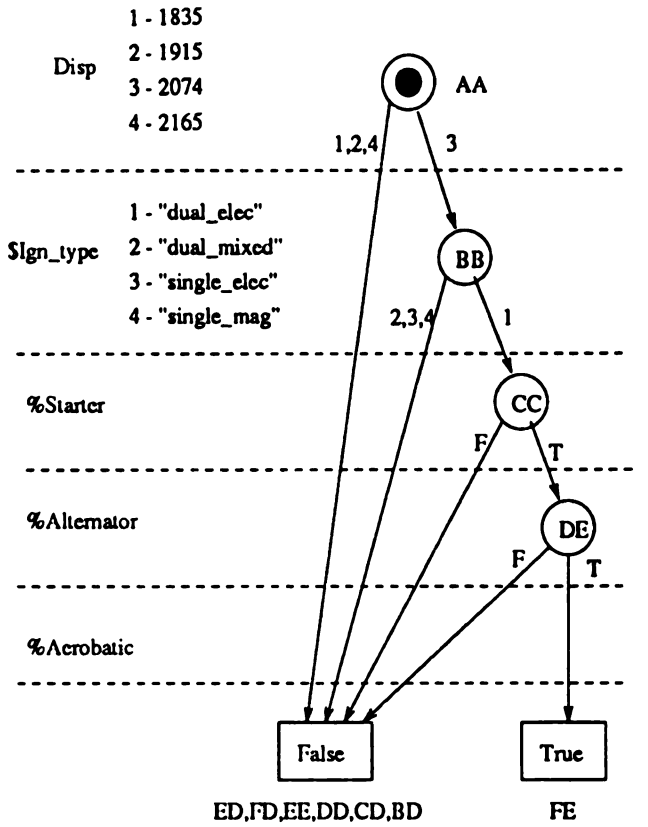


Figure 3.7. MDD after eliminating redundant nodes

node using the recurrence equation $S = L_1 * S_1 + L_2 * S_2 + ... + L_n * S_n$ where L_i is a string characterizing an outgoing arc in terms of the corresponding option variable and S_i is a string characterizing the destination node of that arc. The strings produced are often longer and more complex than what a human would produce. For example, the string produced by this method for the MDD in Figure 3.4 is `(Disp<=1915) * (%Starter * (%Alternator) + %Starter') + (Disp>=2074) * (($Ign_type={'dual_elec','dual_mixed'}) * (%Starter * (%Alternator) + %Starter'))`.

Part III

Framework Services

CHAPTER 4

Computing Property Values

The real power of integrating diverse representations into a common data model is only achieved if interdisciplinary communication is enhanced. In interdisciplinary work, it is important that an engineer from one discipline be able to extract information from representations used by other disciplines. It should not be necessary to learn all of the representations and application tools from other disciplines in order to get needed information.

The message handling system of the proposed framework partially addresses this issue. However, an engineer would still need to know what message to send to the representation in order to extract information from it. Also, many computations involve information from several different representations. In this Chapter, a representation type that provides a convenient interface to the message handling system is presented. Section 4.1 discusses the operation of this representation type from the user's perspective. In the following Section, the implementation is discussed.

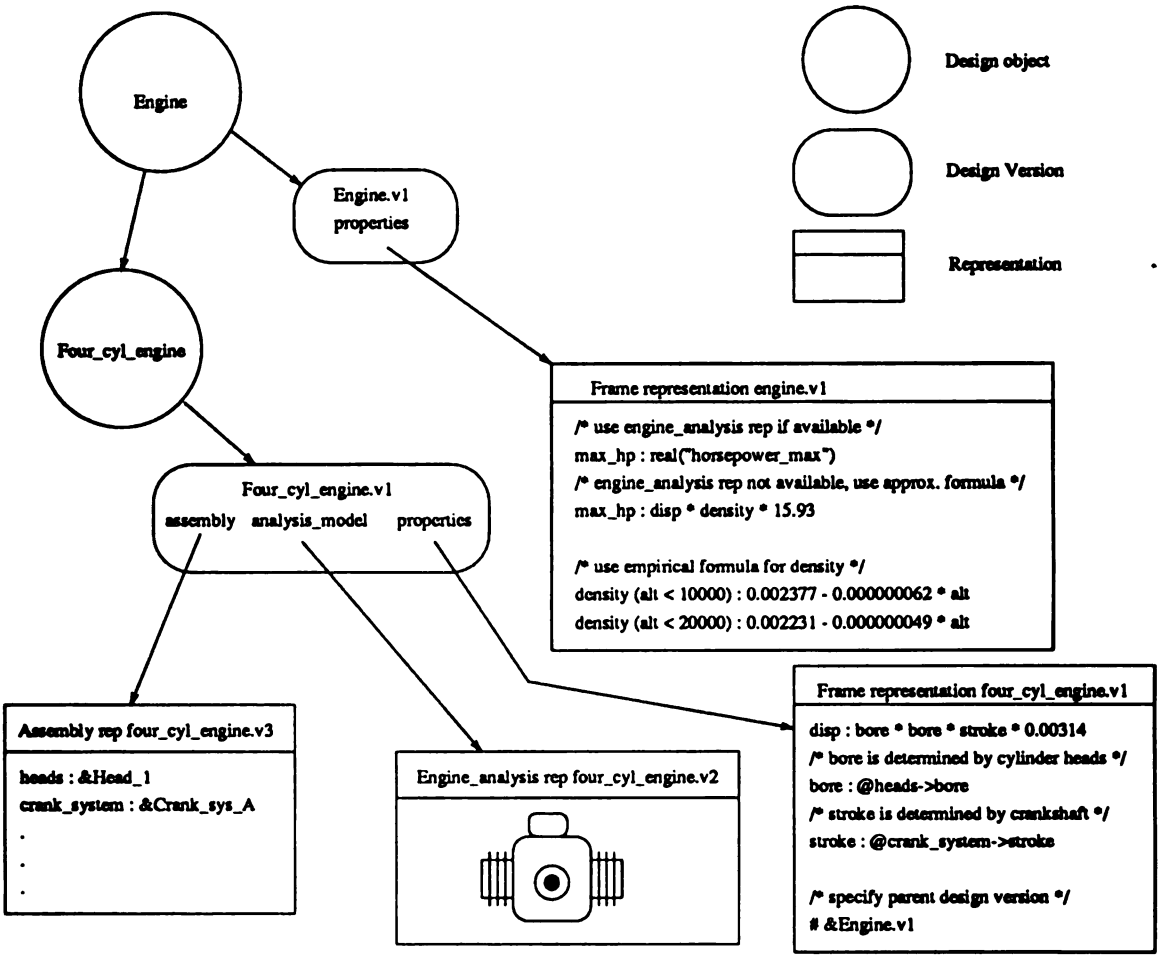


Figure 4.1. Frame representations

4.1 Frame representations

One of the most common requests a designer would make is the computation of a property of a design, such as its weight. A representation type called a `Frame` has been implemented for the primary purpose of responding to "compute <property>" queries. Frames are based on a knowledge representation structure from artificial intelligence [34]. Two frames are illustrated in Figure 4.1. Most of the statements in a frame are of the form `<property name> <applicable condition> : <formula>` indicating a possible formula for computing the named property.

Frequently, there are different formulas that can be used to compute a given property. Which one to use depends on what information is available. For example, if there is a representation available that can respond to the message "horsepower_max", that would probably be the best way to compute the property max_hp. If no such representation is available, a value could be computed using the displacement and the air density. Frames are organized to allow for multiple formulas. A user can enter several different formulas for any variable. The first one that can be successfully evaluated is used.

Some formulas, especially empirical formulas, are only applicable under certain conditions, so the user is allowed to enter a Boolean expression indicating when each formula is valid. When a query arrives for that variable, the Boolean expression is evaluated. If it is false or cannot be evaluated, the formula is not used. If no condition is entered, the formula is assumed to be valid all the time.

General formulas that yield approximate answers with few known parameters are very useful during the early stages of design. For example, the aircraft engine company may have developed empirical formulas, based on previous designs, for the weight of a piston which is a function of bore and stroke only. This formula can be used before the precise geometry and materials have been determined. These general formulas would be placed in frame representations under design objects that are non-leaves in the classification hierarchy. Statements of the form # <binding> forward the message to the referenced design version before processing the remaining formulas. These statements would usually appear at the end of frame representation to access the more general formulas if no formula specific to that design is found.

Figure 4.1 shows typical frame representations. When the frame for four_cyl_engine receives the query "compute max_hp", it searches for an applicable formula. Since no formula is encountered, the message is forwarded to the

design version `Engine.v1`. The first formula it encounters indicates that the message "horsepower_max" is to be sent to the original design version. If there is an `engine_analysis` representation in that design version, it provides the appropriate value. If not, the message fails and the frame tries the next formula, which requires the computation of density and displacement by the original design version. The computation of displacement requires computation of bore and stroke, which are determined by the cylinder head and crank system components, respectively. Computation of these values first requires a query to an `Assembly` representation to determine which design version is used for the component followed by a query to the design version of the component for the value. The empirical formulas for density are only valid over certain altitude ranges. The frame representation uses the first valid formula it encounters.

4.2 Implementation of frame representations

For convenience, frames are implemented as a subclass of textfiles. Thus, they are edited by invoking a standard text editor. The code for computing property values, in response to "compute" messages, is in the `handle_message()` member function of the `frame` class. The algorithm is shown below. Many calculations could involve repeated attempts to evaluate the same property. To avoid repeated computations, the results are stored until the initial message has been handled. The function checks whether the property has already been computed before scanning the formulas. A separate function, `reset_results()`, resets the list of previously computed values each time the user sends a new message. It is assumed that the computations take place quickly enough so that the design data is not modified during the calculation.

```
Algorithm frame_message
```

```
Input:
  A frame representation containing formulas
  The name of a property to compute

Output:
  The property value

1. If this property has already been computed,
   return previously computed result

2. Search through the frame representation line by line {

3.   If the property name matches {

4.     If there was no condition or if it evaluates to true,
       evaluate the formula.

5.     If the formula was evaluated successfully,
       save the result and return the value

     }

6.   If the line is a directive to
     try a different design version {

7.     If binding can be parsed and evaluated,
       send compute message to other design version

8.     If message was handled successfully,
       save the result and return the value

     }

   }
```

CHAPTER 5

Constraint Management

In order to verify that a design is correct, the criteria for correctness must be explicit. These criteria for correctness, called *constraints*, must be managed along with the design data itself. There are a variety of ways of using constraints. The terminology of other authors is used when describing their work.

Schmidtberg and Yerry [4] view constraints as the primary specifications given to component designers by systems designers during top down design.

Buchmann and Perez de Celis [12] elaborate several types of constraints used in chemical process design. The types include checking that a value is within a range or is consistent among objects. In recognition of the need to define constraints dynamically, Buchmann and Perez de Celis express all constraints using strings. They also recognize the need to allow designers to specifically exempt certain designs from certain constraints.

Brown and Breau [35] classify constraints according to how the constraints are used as opposed to how they are expressed. *Inherited* constraints are stored in a classification hierarchy of designs, and express the constraints that must be satisfied by all instances of a design. *In-place* constraints are stored with the design plans (see Part III) and describe conditions that must be satisfied after certain design tasks have been completed. *Implicit* constraints have actually been absorbed into the design

process such that the process guarantees that they are satisfied. Therefore, implicit constraints do not need to be checked. *Accumulation* constraints are similar to in-place or inherited constraints except that they involve properties that are influenced by many different design decisions and therefore must be treated differently.

In his dissertation, Kim describes four categories of constraints [17], which are expressed within VHDL comments. *Performance* constraints restrict some measure of performance of a design, such as area or delay. *Environment* constraints restrict the properties of the designs that can instantiate a design as a component, such as fanout restrictions and operating temperatures. VHDL functions are employed to compute any property values necessary to determine if performance or environment constraints are satisfied. *Relativity* constraints restrict what other designs can be used in conjunction with a design when it is instantiated as a component. *Selection* constraints restrict what designs can be instantiated for a particular component of a design.

A powerful language for expressing constraints is proposed. Section 5.1 discusses how this language is used to express the correctness criteria for designs. Once the constraints are expressed, the framework can help the designer verify that the constraints are satisfied. Section 5.2 discusses how the proposed framework checks the constraints for all variants of a design. It is also possible to make use of the constraints in synthesizing a design, as is discussed in the next chapter.

5.1 Documenting constraints

In the proposed framework, designers can express a wide array of constraints using Boolean expressions. These Boolean expressions are associated with design objects, so they would be considered either inherited or accumulation constraints according to Brown and Breau's classifications [35]. The language allows designers to express

many different types of constraints. For example, the following constraints may apply to the design of the cylinder head assembly (the language syntax is described in Appendix B):

`(weight <= 12)` specifies a maximum acceptable weight.

`(\->stroke = [69,78])` specifies that this assembly is only usable on an engine with a stroke between 69 and 78 mm.

`(~geom->diameter = ~process->diameter)` specifies that the diameter according to the representation describing the manufacturing process is the same as the diameter according to the representation describing the geometry.

`(@Head->IDiam = @Cylinder->bore)` specifies that the inside diameter of the cylinder and cylinder head must be equal.

`(~process->string('check buildable') = 'ok')` specifies that the design must be manufacturable, which is checked by sending a message to the representation describing the manufacturing process.

The degree to which a certain design version satisfies the constraints is a measure of the designs maturity, and is therefore related to release control. Managers may wish to insist that all constraints be satisfied before a design version may be released into a workspace that is widely shared. However, the constraints may be relaxed for workspaces that are intended for designs in progress. For example, the engineers working on parts of the cylinder head assembly may use a certain workspace, called `Cyl_Head_Prelim`, for sharing semi-validated designs among themselves. This workspace would be a child of the workspace that contains highly validated designs for company wide sharing. The individual engineers would create their own workspaces as children of `Cyl_Head_Prelim` to work on experimental designs of each part. A manager may decide to relax the constraint on maximum weight for releasing new

cylinder head design objects into the `Cyl_Head_Prelim` workspace. Presumably, the weight will be reduced by further design refinement before the design is released into the parent workspace for wider sharing. Some type of inheritance of constraints within workspaces is needed. Normal inheritance would produce the opposite of the desired affect;1 child workspaces would be more constrained than parent workspaces. To get the desired relationship between constraints and workspaces, designers may exempt workspaces from certain constraints. These exemptions are inherited by children workspaces.

5.2 Constraint checking

When a designer feels that a design version is correct, he verifies it by checking the constraints. Constraints are evaluated the same way as other Boolean expressions are evaluated. Specifically, when the user asks to evaluate the constraints, the Boolean expression is evaluated using whatever workspace and option selection are currently in force. For example, to evaluate the constraint on maximum weight, a message would be sent to the design version to compute the weight. A frame representation would handle the message, perhaps by sending messages to each component to compute their weight and adding up the results. This result is compared to the constant 12 to determine if the constraint is satisfied.

However, constraints containing references to the environment, called *environment constraints**, cannot be evaluated until the design is instantiated as a component in another design. Therefore, environment constraints must be treated separately. To validate a design, the designer must check all of the non-environment constraints of the design and all of the environment constraints of the components.

*This use of the term environment constraint is consistent with its use in [17].

The design must be validated for its entire applicability set. However, it is often not possible to evaluate the expressions that define the constraints without specifying some of the options. For example, if the choice of design for some of the components depends upon the option variable Bore, then the weight could not be computed. Two algorithms are proposed to check the constraints for the entire applicability set. The first algorithm uses a primitive approach to divide the applicability set into subsets such that the constraints can be evaluated directly for each subset. The second algorithm utilizes the parse tree of the expression to build an MDD for the expression.

When the constraints cannot be checked for the entire applicability set due to resource limitations, it can be checked for only those option combinations defined by a particular option selection object. Checking only some option combinations involves some risk that a constraint would be violated by another combination. When all combinations cannot be checked, it may be best to wait until the design is instantiated as a component in another design (or ordered by a customer) and then check it only for the combinations that actually get used.

The basic idea of the first algorithm is to divide the applicability set into disjoint subsets such that the expression can be evaluated for each subset. There are many different ways to divide the applicability set. The easiest way is to pick an option variable and divide the applicability set according to the value of that variable. In the cylinder head example, all of the option combinations with Bore = 92 might be placed in one set and those with Bore = 94 in another set. Given an MDD enumerating the entire applicability set, it is simple to construct an MDD enumerating each of the subsets. It is likely that it may still be impossible to evaluate the expression for some of the subsets in which case another option variable must be chosen and the subset divided further. The sets are repeatedly divided until the constraint can be evaluated or the set contains a single option combination.

Once the system has evaluated a constraint, it reports to the user the number of combinations for which the constraint is satisfied and the number of combinations for which it fails. The user may choose to list only some of the option variables. If the list of option variable has been exhausted and it is still impossible to evaluate the expression, the system simply reports those cases as "unknown". Users can also request that the system print out one of the option combinations that failed or could not be evaluated. This information about which option combinations failed is very helpful in determining how to modify the design to satisfy the constraints.

Both algorithms are more general than they need to be for evaluating constraints, because they can evaluate numeric and string valued expressions in addition to Boolean expressions. The reasons for this generality are be discussed when the second algorithm is presented.

```
Algorithm check_constraint_A

Input:
  A Boolean, numeric, or string valued expression, E
  An MDD, S_in,  representing a set option combinations
  An ordered list of option variables, L

Output:
  MDD S_out representing the result of evaluating the
    expression for the combinations represented by S_in

1.  Attempt to evaluate E directly for all
    of the set of combination represented by S_in.

2.  If E evaluates to X, return S_out = X for
    combinations in S_in, unknown for others.

3.  If evaluation fails {

4.    If L is empty, return S_out = unknown.

5.    Remove option variable V from the beginning of L

6.    Produce new MDDs S_in[1]...S_in[n] by restricting S_in
      to combinations with each value of V.
```

```
7.    Call Algorithm check_constraint_A recursively with
      S_in[1]...S_in[n] to determine S_out[1]...S_out[n].
      L contains one fewer option variables in recursive call.

8.    Compute S_out = by combining S_out[1]...S_out[n]
      using Algorithm node_combine of Chapter 3.

   }
```

Theorem 5.1 states that, if L contains all of the option variables, *S_out* will evaluate to the same value as the expression for any combination of option values. However, if some option values are not included in L, then *S_out* may not assert a value (evaluate to unknown) for some option combinations. It will never incorrectly assert a value. In order for the theorem to hold, an assumption must be made about the expression evaluation algorithm called in step 1.

Assumption 1 *If an expression evaluates to X for a set of option combinations, then it also evaluates to X for each individual option combination in that set.*

Theorem 5.1 *If Assumption 1 holds and L contains all of the option variables, then for any combination of option values C within the set represented by S_in, the MDD S_out produced by Algorithm check_constraint_A will evaluate to the same value as E.*

Proof: If the evaluation in step 1 is successful, Assumption 1 guarantees that step 2 sets *S_out* such that the theorem holds. If the evaluation in step 1 fails, the algorithm divides the set of combinations into progressively smaller subsets such that C is included in exactly one of the subsets. After all of the variables have been used to split *S_in* into subsets, each subset will contain a single combination. If the expression can be evaluated for C, then the algorithm will eventually succeed in evaluating a subset containing C and set *S_out* such that it evaluated to the same value for C If the expression cannot be evaluated for C, then the algorithm will set *S_out* such that

it evaluates to unknown for C. □

In the worst case, this algorithm ends up evaluating the constraint for each option combination individually, making the complexity exponential in the number of option variables. This complexity should not be surprising because the constraint checking problem is NP-complete.[†] In many cases, however, it is able to evaluate many combinations at once because some option variables do not enter into the expression. Unfortunately, the order in which option variables are used to divide the applicability set has a profound effect on performance. It is not easy to determine an appropriate order automatically. In the current implementation, the user must specify the sequence of option variables to use in dividing the applicability set.[‡]

A drawback of Algorithm check_constraint_A is that a subexpression that does not depend on some of the option variables might be evaluated many times. Consider the expression $(@Head- > weight + @Cylinder- > weight < 10)$. The subexpression $@Head- > weight$ might not depend on `Stroke` while $@Cylinder- > weight$ does. Under Algorithm check_constraint_A, both subexpressions must be re-evaluated for the subsets defined by `Stroke`.

The next algorithm avoids unnecessary subexpression evaluations by utilizing the expression's parse tree. First, MDDs are constructed for each leaf in the expression's parse tree using Algorithm check_constraint_A. Then, these MDDs are combined using Algorithm node_combine from Chapter 3. Algorithm check_constraint_B is potentially much more efficient because most of the leaf subexpressions may depend on few option variables whereas the expression as a whole may depend on several more.

[†]The constraint is a Boolean function of the option variables. Checking that the constraint is satisfied is the same as testing satisfiability of its complement function. Option variables are more general than Boolean variables. Therefore the Boolean satisfiability problem, a well known NP-complete problem, is a special case of this problem.

[‡]The order used to divide the applicability set may be different than the option variable ordering of the MDD.

Algorithm check_constraint_B is complicated by the fact that the appropriate sequence of option variables to send to Algorithm check_constraint_A may be different for each leaf subexpression. The language syntax can be modified such that certain types of expressions optionally contain a list of variables to assist the framework in building the MDD. These lists of variables do not influence the value of the expression, but do influence the resources needed to check constraints.

Another enhancement improves the efficiency for subexpressions that access a property of a component design. For these subexpressions, using the case labels of the generalized binding to subdivide the set of option combinations is likely to be more effective than using an option variable list. The enhanced algorithm is presented below:

```
Algorithm check_constraint_B

Input:
  A Boolean, numeric, or string valued expression, E
  An MDD, S_in, representing a set of option combinations
  An ordered list of option variables, L

Determine:
  MDD S_out representing the result of evaluating the
    expression for the combinations represented by S_in

1.  If expression E is formed by combining subexpressions
    with an operator {

2.    Compute S_1 and S_2 for each subexpression, E_1 and E_2,
      using Algorithm check_constraint_B
      (arguments E_1 or E_2, S_in, and L)

3.    Compute S_out by combining S_1 and S_2
      using Algorithm node_combine of Chapter 3

    }

5.  If expression E is a component property {

6.    If the component uses a static or dynamic binding,
```

```
        compute S_out using Algorithm check_constraint_A
        (arguments E, S_in, and L)

7.      If the component uses a generalized binding {

8.        Produce new MDDs S_in[1]..S_in[n] by restricting S_in
          according to the case labels

9.        Compute S_out[1]...S_out[n] using Algorithm
          check_constraint_A for S_in[1]...S_in[n]
          (arguments E, S_in[i], and L)

10.       Compute S_out by combining S_out[1]...S_out[n]
          using Algorithm node_combine of Chapter 3

        }

11.   Otherwise, compute S_out using Algorithm check_constraint_A
      (arguments E, S_in, and L concatenated to expression's
      list of option variables)

}
```

Theorem 5.2, which is similar to Theorem 5.1, states that, if L contains all of the option variables, *S_out* will evaluate to the same value as the expression for any combination of option values.

Theorem 5.2 *If Assumption 1 holds and L contains all of the option variables, then for any combination of option values C within the set represented by S_in, the MDD S_out produced by Algorithm check_constraint_B will evaluate to the same value as E.*

Proof: Let i denote the number of levels in the expression's parse tree. The proof is an induction on i. As an induction basis, consider only leaf subexpressions ($i = 0$). The proof of the basis is separated into three cases:

Case 1 - Component properties with static or dynamic bindings This case is handled in step 6, and is covered by Theorem 5.1.

Case 2 - Component properties with generalized bindings This case is handled by steps 7-10, which are similar to steps 5-8 of Algorithm check_constraint_A. The only difference is the method used to divide the set of combinations into subsets. The proof of this case is similar to the proof of Theorem 5.1.

Case 3 - Other leaf expressions This case is handled in step 11, and is covered by Theorem 5.1.

Let us examine the result of evaluating an expression with an $i+1$ level parse tree. The induction hypothesis guarantees that the MDDs computed in step 2 obey the theorem. Algorithm node_combine (called in step 3) guarantees that it holds for the combined expression. □

5.3 Limitations

One drawback of the proposed functionality is that it is very difficult to know when constraints must be re-checked. Even if the design itself is not modified, changes to components could cause some of the constraints to be violated. Change notification functionality, which is discussed in Chapter 7, helps identify situations in which constraints should be re-checked.

Although the constraint checking discussed above is a useful design tool, it does not support a truly constraint driven design process. In a constraint driven process, a designer would not look at the details of the component's designs, but would instead impose sufficient constraints upon the components to guarantee satisfaction of the constraints imposed upon his design. For example, when checking that the maximum weight is not exceeded, the designer would not use the exact weight of each component, but rather the weight limit of each component as indicated in the component's

constraints. That way, it does not matter what changes are made to the component's design as long as the components satisfy their constraints. Checking the constraints using only the constraints of components is a much more difficult problem.

CHAPTER 6

Automated Component Selection

This chapter is a slight departure from the previous chapters in that it describes algorithms for actually performing design as opposed to supporting design indirectly. These algorithms would be implemented as tools which would be tightly integrated into the framework.

For many design problems, the general arrangement of parts has already been determined and the designer must select a design for each part. This type of design problem is what Navathe describes as "design by selection" [5] *. For example, the designer of the cylinder head assembly may know that he needs a cylinder, a cylinder head, spark plugs, and a rocker box assembly, but several of each are available. The structure of this design would be described by an assembly representation with bindings to design objects which are not leaves in the classification hierarchy. To further specify the design, a specific design should be chosen for each component and any options of those designs must be specified. Programs which can help the designer find an acceptable combination of component designs would be very useful.

*Some authors refer to this type of design as "configuration design" or "configuration binding" [17]. However, the term "configuration" has a different meaning in this dissertation (see Chapter 2).

As observed by Kott and May [36], some products can be decomposed into nearly independent components while others exhibit strong interactions between components. When the product is *nearly decomposable*, the general structure of the design is fixed a priori, at least to a small number of alternatives. The main task of the designer is to select among those alternatives and determine what design to use for each component. When the interactions are strong, the main task is to determine the arrangement of parts. This chapter describes framework support for the former type of problem.

The basic component selection problem is defined first. Then, two more interesting variations of the component selection problem are defined: the component selection problem with options, and the recursive component selection problem. Known solutions to each of the variations are discussed. New solutions are proposed for the basic component selection problem and the component selection problem with options. The proposed algorithms are very general, but are inefficient under certain circumstances, which are characterized.

6.1 The basic component selection problem

The basic component selection problem is defined as given:

- a design that specifies the arrangement of components, but not the design for each component,
- a set of candidate designs for each component,
- a set of constraints, and
- an objective function,

determine which design should be used for each component such that the constraints are satisfied and the objective function is optimized.

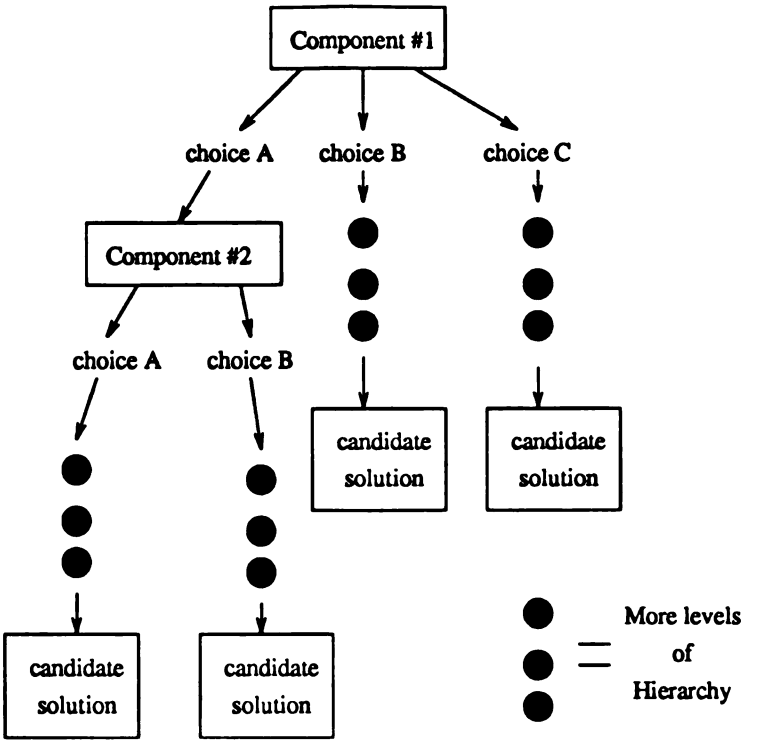


Figure 6.1. A component selection search tree

The component selection problem is NP-complete, but may be solved practically in many cases, especially when the objective function is omitted (any solution satisfying the constraints is acceptable). Most previous algorithms employ a branch and bound strategy. Branch and bound algorithms utilize a search tree as illustrated in Figure 6.1. In order to avoid searching the entire tree, a rating is calculated for interior nodes estimating the likelihood of meeting all constraints (or estimating the objective function), based on the components that have been selected to that point. The most promising paths are searched first. However, in many cases, such rating functions do not exist.

A unique solution, employing a genetic algorithm, is presented by Brown and Hwang [37]. Each candidate solution is represented by a binary string. An initial population of candidates is generated randomly. Each candidate is rated according

to how well it satisfies constraints. New candidates are generated by selecting two existing candidates such that highly rated candidates are more likely to be picked, breaking each string at a randomly selected point, and combining the first part of one with the last part of the other. During each *generation*, new candidates are added to the population and low rated candidates are dropped from the population. Also, a small percentage of the candidates are *mutated* by randomly switching a few bits. The authors claim that for the domains they have investigated, this approach finds acceptable solutions much faster than branch and bound algorithms for large search spaces.

Kim [17] describes an algorithm for solving a special case of this problem where the design is a combinational circuit and the constraints are restrictions on the maximum delay and the total area. Kim's algorithm is efficient when the circuit is mostly series-parallel.

In the proposed algorithm, component selection is re-formulated as a constraint problem. For each component, a new option variable is introduced with one value for each possible choice. These new option variables are called *supplemental* option variables. The supplemental option variables are discarded after the components are selected. The bindings for unspecified components are temporarily replaced with generalized bindings which use these supplemental option variables in the case labels. An MDD is created for the conjunction of the constraints. This MDD provides the designer with important information about which combinations of component selections satisfy the constraints. Specifically, one combination of values of supplemental option variables which satisfies the constraints and optimizes the objective function must be identified. The process is presented algorithmically below and illustrated with an example.

In what situations is the proposed procedure be effective? If there are many components and many candidate designs for each, then the procedure investigates an

extremely large search space. The previous observation about the efficiency of MDDs for processing option restrictions relied on the fact that most option restrictions only involve a small number of option variables. Similarly, MDDs are effective for this problem when most of the constraints involve only one or a small number of option variables. Fortunately, as noted by Kott et. al. [38], most constraints in many disciplines only apply to a single component. Others constraints check for compatibility between two components. Very few involve all or most of the components.

Certainly, designers need to use this procedure carefully, with an understanding of its limitations. In many cases, they may be able to create bindings for many of the components themselves and only rely on the procedure to select a few components. Also, designers should specify very simple objective functions whenever possible. If the objective function has many different values, its MDD is very large. Still, there are many situations where this procedure saves a designer considerable time relative to manual exploration of the search space.

The overall algorithm is described as follows:

```
Algorithm select_components

Input:
  A design with several undetermined components
  For each undetermined component, a list of candidate designs
  A set of constraints
  An objective function

Output:
  A binding for each component indicating the design
    to be used such that:
      - all constraints are satisfied.
      - the objective function, if given, is optimized.

1.  Create the supplemental option variables and temporary
    bindings.

2.  Compute the selection MDD

3.  Determine the bindings for each component
```

The first step, setting up the supplemental option variables and temporary bindings proceeds as follows:

```
Procedure setup_basic
/* Create the supplemental option variables and
   temporary bindings. */

1. For each component {

2.   Create a supplemental option variable.

3.   Create a temporary generalized binding for the component.

4.   For each candidate design {

5.     add a possible value to the component's supplemental
       option variable.

6.     Add a case to the components binding which binds to the
       candidate whenever the supplemental option variable takes
       on the new value.

     }
  }
```

The next step is computation of the selection MDD. The simplest constraints, such as those that involve only a single component, are processed first. In that way, the more complicated constraints are evaluated for fewer combinations. Any path through this MDD to a terminal TRUE node represents a combination of component designs that satisfy the constraints. Multiple possibilities manifest themselves in two ways: non-terminal nodes with more than one outgoing edge to non-FALSE nodes and edges that skip levels. The objective function is used to select one combination from those that satisfy the constraints. Algorithmically,

```
Procedure compute_selectionMDD_basic
/* Compute the selection MDD */
```

```
1. Rank the constraints according to degree of complexity.

2. Let S be an MDD for the Boolean expression TRUE.

3. For each constraint, C, from simplest to most complicated {

4.    Compute the MDD S' which satisfies C using Algorithm
      check_constraint_A or check_constraint_B with S_in = S.

5.    Let S = S * S', computed using algorithm node_combine
      (S now represents the combinations that satisfy all
      constraints processed so far)

   }

6. Compute MDD S_sel for the objective function,
   restricted to the combinations represented by S using
   algorithm check_constraint_A or check_constraint_B.

7. For any edge that skips a level in S_sel, add non-terminal
   nodes at each skipped level with one arbitrarily chosen edge
   to the node at the next level.
```

Once the selection MDD has been computed, the bindings can be determined as follows:

```
Procedure compute_bindings_basic
/* Determine the Bindings for each Component */

1. Choose one path through the selection MDD, S_sel, that
   optimizes the objective function.

2. For each node in this path {

3.    The binding for the component corresponding to this
      node's option variable is a dynamic binding to the design
      corresponding to the outgoing arc on the selected path.

   }
```

Theorem 6.1 states that algorithm select_components selects the optimum combination of components that satisfy all of the constraints. The proof relies on the

following lemmas which state properties of the selection MDD produced in procedure compute_selectionMDD.

Lemma 6.1 *If a combination of candidate components fails a constraint C, then S_{sel} does not assert a value (evaluate to false) for the corresponding combination of supplemental option values.*

Proof: When constraint C is processed in step 4 of Procedure compute_selectionMDD_basic, the combination may or may not be included in the previous MDD S. If it is included, Theorems 5.1 and ?? guarantee that that combination evaluates to false in S'. After step 5, S evaluates to false for that combination either way. When additional constraints are processed, step 5 always computes S such that it evaluates to false for the combination that failed C. Step 6 sets S_{sel} such that it evaluates to unknown for the combination that failed constraint C. Step 7 never causes S_{sel} to assert a value for any combination which previously evaluated to unknown. □

Lemma 6.2 *If the design which uses a combination of candidate components satisfies all constraints and the objective function for that design evaluates to X, then S_{sel} evaluates to X for the corresponding combination of option values.*

Proof: If the constraints all evaluate to true, then S evaluates to true for that combination in step 6, as described above. Theorems 5.1 and ?? guarantee that S_{sel} evaluates to the correct value after step 6. □

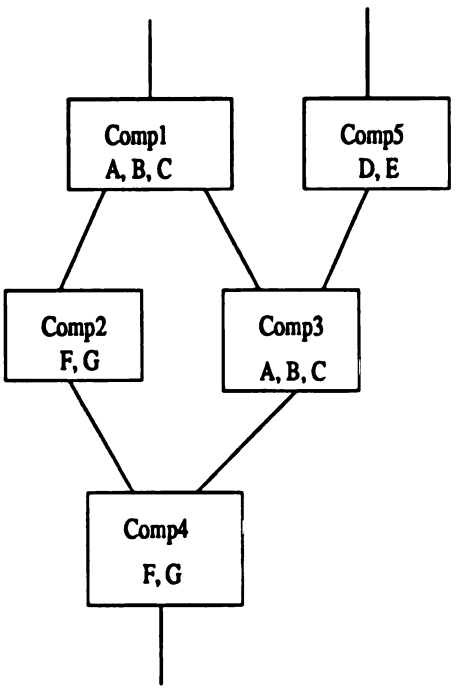
Theorem 6.1 *The components selected by Algorithm select_components satisfy all of the constraints. Furthermore, no other combination of components (among those listed as candidates) satisfies all of the constraints and evaluates to a superior value of the objective function.*

Proof: Each of the new bindings evaluates to the same design version as the temporary binding with the option variables set to the selected components. Only paths through S_{sel} which end in a numeric terminal node are considered by step 1 of procedure compute_bindings_basic. If the design failed a constraint, Lemma 6.1 states that the corresponding path through S_{sel} would not lead to a numeric terminal node. If some other combination of components satisfied all constraints and resulted in a superior value of the objective function, Lemma 6.2 guarantees that there would be a path through S_{sel} leading to a terminal node with the superior value. That path would have been selected in step 1 of procedure compute_bindings_basic. □

Suppose that a digital circuit designer has determined that a circuit should be composed of five components, arranged as illustrated in Figure 6.2. For each component, there are either two or three alternatives designs, with area and delay properties as indicated. The circuit has constraints that the total area of the five components must be less than 55 and the maximum delay must be less than 36. The objective is to determine which design should be used for each component such that the constraints are satisfied and the least power is required. Note that these constraints all involve several components, violating one of the conditions for efficiency of the proposed algorithm. This example was chosen to illustrate the generality of the algorithm.

To solve this problem, the setup procedure creates one string valued supplemental option variables for each component $C1, $C2, $C3, $C4, and $C5. Each variable has a set of possible values indicating what designs may be used, for example $C2 has legal values "F" and "G". These option variables are used in generalized bindings for each component. The binding for Comp2 would be: "`for {($C2 = 'F') use &F; ($C2 = 'G') use &G;}`". These option variables allow 72 different combinations.

In the next phase, the MDD in Figure 6.3 is constructed by evaluating the constraints using one of the algorithms of the previous chapter. In this case, there are 5



Schematic

A	area = 10	delay = 10	power = 9
B	area = 5	delay = 12	power = 10
C	area = 15	delay = 8	power = 11
D	area = 15	delay = 10	power = 3
E	area = 10	delay = 12	power = 4
F	area = 10	delay = 12	power = 5
G	area = 5	delay = 14	power = 5

Component Choices

Figure 6.2. Digital Circuit Design Problem

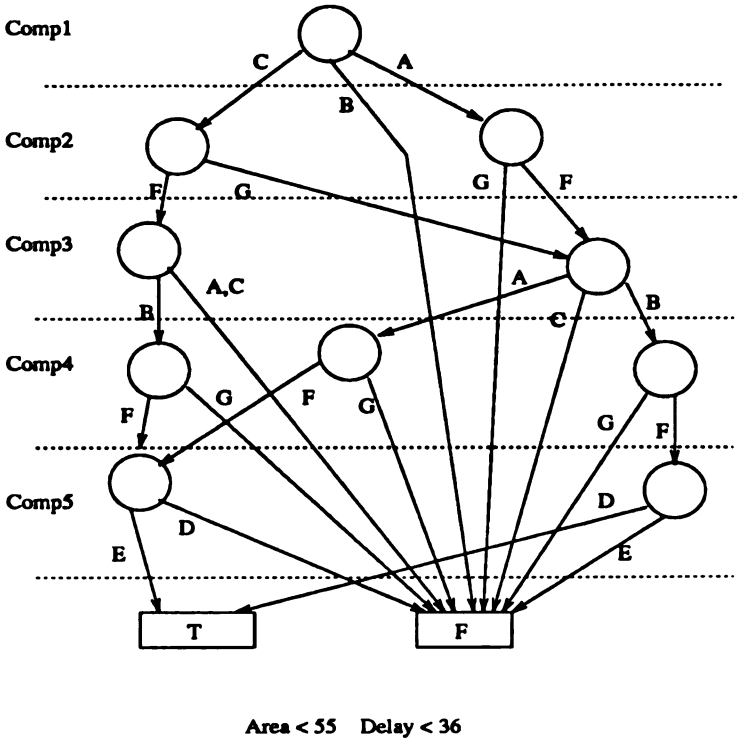


Figure 6.3. MDD for Combined Constraints

different combinations which satisfy all of the constraints. The selection MDD, which represents the objective function restricted to the combinations that satisfy the constraints, is shown in Figure 6.4. Multiple combinations which satisfy the constraints results in non-terminal nodes with more than one outgoing arc or in edges that skip a level. In the final phase, `A` for `Comp1`, `F` for `Comp2`, `B` for `Comp3`, `F` for `Comp4`, and `D` for `Comp5` are chosen arbitrarily from among those combinations that minimize the objective function.

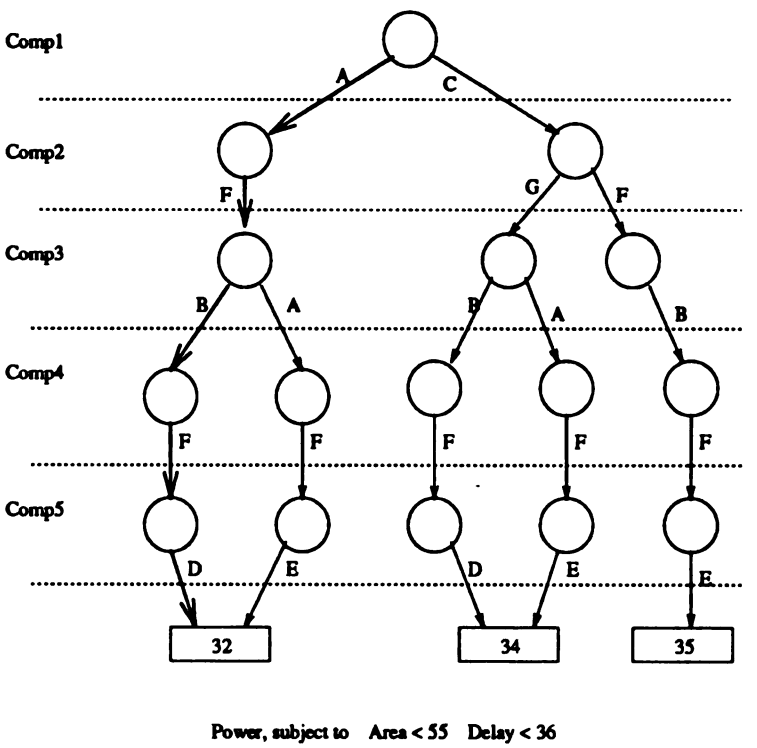


Figure 6.4. MDD for Objective Function

6.2 The component selection problem with options

Recall that a product family is a design that offers an array of options and a variant is a particular member of the product family. The use of product families with many variants complicates the component selection problem. In the component selection problem with options, the design and the candidate component designs each have many variants. The component designs, including option values, must be selected for all variants of the top level design.

In [27], Blaha et. al. describe a system which analyzes previous variants and automatically develops rules for component selection. The population of variants must initially be created manually by designers. There is no guarantee that a given population of variants generates rules that are unambiguous and consistent.

Malmqvist [39] discusses a system which assists the user in designing variants using information stored about the product family and the component families. For each product family, the set of components are stored in a database, but only the component family is specified. The particular component variant is chosen later when a product variant is designed. The user specifies parameters of the desired product variant (choices for optional features), and the system calls programs that design variants of each component.

Since most variants differ only slightly, it may be possible to avoid determining the components for each variant independently. Also, when a candidate component design is an entire family, treating each variant of that family as a separate candidate should be avoided. The basic steps in the algorithm are the same. However, some of the details must be modified as shown below. Due to the large number of candidate variants that are likely to be available, the search space can be enormous. However,

this version of the algorithm is efficient under the same circumstances as the proposed algorithm for the basic component selection problem.

Additional steps must be added to the procedure which creates supplemental option variables and temporary bindings to handle candidate component designs with options. For each option variable of the candidate component, a supplemental option variable is created with the same choices. If two or more of the candidates inherit a common option variable, only one supplemental option variable is needed. The option values for the component are specified in the binding, as described in Chapter 3. In the temporary bindings, the option value of the component is set equal to the corresponding supplemental option variable.

```
Procedure setup_with_options
/* Create the supplemental option variables and
   temporary bindings. */

1. For each component {

2.   Create a supplemental option variable.

3.   Create a temporary generalized binding for the component.

4.   For each candidate design {

5.     Add a possible value to the components supplemental
       option variable.

6.     Add a case to the components binding which binds to the
       candidate whenever the supplemental option variable takes
       on the new value.

7.     Add a new supplemental option variable, V', corresponding
       to each option variable, V, of the candidate design.
       (If another candidate has the same option variables,
       they can share supplemental option variables.)

8.     Add string expressions to the binding referencing
       the candidate that equates V to V'.

  }
```

```
}
```

The supplemental option variables always follow the regular option variables in the selection MDD. The region of the selection MDD corresponding to supplemental option variables is called the *supplemental region*, while the region corresponding to regular option variables is called the *primary region*. The procedure for creating the selection MDD is modified as shown below. In step 2, S is set to the applicability set instead of the the Boolean expression TRUE. Steps 8 and 9 are added to remove ambiguity when more than one combination of components satisfies the constraints.

```
Procedure compute_selectionMDD_with_options
/* Compute the selection MDD */

1. Rank the constraints according to degree of complexity.

2. Let S be an MDD for the applicability set.

3. For each constraint, C, from simplest to most complicated {

4.   Compute the MDD S' which satisfies C using Algorithm
     check_constraint_A or check_constraint_B with S_in = S.

5.   Let S = S * S', computed using Algorithm node_combine
     (S now represents the combinations that satisfy all
     constraints processed so far)

   }

6. Compute MDD S_sel for the objective function,
   restricted to the combinations represented by S using
   Algorithm check_constraint_A or check_constraint_B.

7. For any edge that skips a level in S_sel, add non-terminal
   nodes at each skipped level with one arbitrarily chosen edge
   to the node at the next level.

8. Working from the bottom up, assign a value to all
   non-terminal nodes in the supplemental region using the
   recursive formula: The value of a non-terminal node is the
   maximum of the values of any of its descendants.
```

```
9. For any non-terminal node N in the supplemental region,
   remove all outgoing edges except one to a descendant
   node N' with the same value as N.
```

The procedure to determine the binding for each component is completely changed, as shown below:

```
Procedure compute_bindings_with_options
/* Determine the Bindings for each Component */

1. For each component in the original design {

2.   Create a generalized binding for the component.

3.   For each candidate design {

4.     Build a new MDD, A, by restricting S_sel to the
       combinations for which the components supplemental
       option variable takes on the value corresponding
       to this candidate.

5.     Replace all of the non-terminal nodes in the supplemental
       region of A by terminal true nodes and reduce A.
       (The result represents the option combinations for
       which the candidate should be used for the component.)

6.     If this MDD is not empty {

7.       Create a string for the Boolean expression that is
         equivalent to A. This will be the case label
         for one case of the generalized binding.

8.       The binding will be a dynamic binding, with string
         expressions determined from the selection MDD
         as follows {

9.         For each option variable V  {

10.          For each potential value of that variable X {

11.            Build a new MDD, B, by restricting S_sel to the
               combinations in which V' = X. Replace
```

```
                non-terminals in the supplemental region by
                terminal true nodes and reduce. (B describes the
                option combinations that require choice X for V.)

12.             If B is not empty, convert it to a string, C.
                Add the string expression
                :C: + -> (option = value) to the dynamic binding.

            }
          }
        }
      }
    }
  }
```

Theorem 6.2 states that this algorithm results in a properly specified design. Theorem 6.3 states that the design produced satisfies the constraints and optimizes the objective function.

Theorem 6.2 *If some combination of components satisfies the constraints for a particular variant, then the bindings produced by procedure compute_bindings_with_options specify a design for each component.*

Proof: Notice that steps 7 through 9 of procedure compute_selectionMDD_with_options ensure that:

- Starting at any edge that crosses the boundary between the primary region and the supplemental region, there is exactly one path to a numeric terminal node (no branching).
- All of these paths have one node for each supplemental option variable (no levels are skipped).

Every variant of the design corresponds to an edge which crosses into the supplemental region. The path starting with that edge indicates the components for that variant.

Suppose that candidate design D_i is to be used for the component corresponding to supplemental option variable V_i of a particular variant K. K's path through the supplemental region has one node corresponding to V_i with one outgoing edge corresponding to D_i. MDD A, produced in step 4 of procedure compute_bindings_with_options, includes this path and all of the nodes in the primary region which lead to this path (see Figure 6.7). In step 5, the supplemental region is removed from A such that A evaluates to true for the option combination that defines K (see Figure 6.8). Step 7 creates a case label which evaluates to true for K. None of the case labels for other candidates for this component evaluate to true for K.

Steps 9 through 12 create the string expressions that specify which variant of D_i to use. By similar arguments to those above, exactly one of the strings C in step 12 evaluates to true for K for each option variable V. Therefore, these string expressions assign a value to each option variable of D_i, uniquely identifying one variant of D_i for use in K. □

Theorem 6.3 *For each variant, the components selected by Algorithm select_components satisfy all of the constraints. Furthermore, no other combination of components (among those listed as candidates) satisfies all of the constraints and evaluates to a superior value of the objective function.*

Proof: The proof is similar to the proof of Theorem 6.1. □

Consider an example in which a designer wishes to select the components for a cylinder head assembly that consists of a cylinder, a head, and either one or two spark plugs. The following options are to be offered:

Disp The displacement can be one of the five values 1679, 1835, 1915, 2074, 2165.

$Ign The following types of ignition systems are offered: 'single_mag', 'single_elec', 'dual_elec', 'true_dual'.

%EMI_safe If the engine will be used near radio equipment it should not generate interference.

The design must satisfy the following constraints:

(@Cylinder->(bore*bore*stroke*3.14)=[Disp-5,Disp+5]) - The actual displacement must be within 5 cc of the specified value.

(@Cylinder->bore=@Head->bore) - The cylinder and head must have the same bore.

($Ign={'single_mag','single_elec'})->(@Head->$ign='single') - If single ignition is ordered, the head should have only one spark plug hole.

($Ign={'dual_elec','true_dual'})->(@Head->$ign ='double') - If dual ignition is ordered, the head should have two spark plug holes.

(%EMI_safe*($Ign={'single_mag','true_dual'}))->@Plug1->%Shielded - If the first plug is fired by a magneto, it must be shielded to avoid radio interference.

(@Plug1->diam=@Head->plug_diam) - The first plug should be the right diameter to fit the hole in the head.

($Ign={'dual_elec','true_dual'})->(@Plug2->diam=@Head->plug_diam) - If needed, the second plug should also be the right diameter to fit the hole in the head.

($Ign={'single_mag','single_elec'})->(@Plug2 =&NULL) - If single ignition is ordered, there should not be a second spark plug.

The following cylinder designs are available:

P10_025 - 88mm bore, 69mm stroke

P10_027 - 92mm bore, 69mm stroke

P10_100 - 94mm bore, 69mm stroke

P10_101 - 94mm bore, 78mm stroke

P10_138 - 92mm bore, 78mm stroke

The following head designs are available:

P10_043 - 88mm bore, single ign, takes 14mm plugs

P10_045 - 92mm bore, single ign, takes 14mm plugs

P10_047 - 92mm bore, dual ign, takes 14mm plugs

P10_146 - 94mm bore, single ign, takes 14mm plugs

P10_147 - 94mm bore, dual ign, takes 14mm plugs

The following spark plug designs are available:

SP1 - 14mm, has option %Shielded

SP2 - 10mm, has option %Shielded

The setup procedure creates a supplemental option variable for each component. In step 7, the procedure also creates the supplemental option variables `%Plug1_Shielded` and `%Plug2_Shielded` because the candidate designs for the spark plugs have an option. The binding for the first spark plug would be: "`for {(Plug1 = 1) use &SP1, '(%Shielded = '+:%Plug1_Shielded:+')'; (Plug1 = 2) use &SP2, '(%Shielded = '+:%Plug1_Shielded:+')';}`". This

binding states that the choice between SP1 and SP2 is determined by the supplemental option variable Plug1 and that the choice for the option %Shielded is determined by the option variable %Plug1_Shielded. Refer to Appendix B for a detailed description of the syntax.

In the second phase, the MDD of Figure 6.5 is produced by evaluating the constraints (steps 1 through 5 of procedure compute_selectionMDD). Arcs that go directly to a terminal false node are omited for clarity. The designer observes from this MDD that there are no components available to construct a head assembly with a displacement of 1679cc and dual ignition. Therefore, he adds a restriction to the design object to preclude ordering a head assembly with those options.

For some variants, it does not matter whether the spark plug is shielded or not. This situation is indicated by arcs which skip the levels of the supplemental option variables %Plug1_Shielded and %Plug2_Shielded. This ambiguity is handled by adding nodes in these level which arbitrarily specify unshielded plugs as shown in Figure 6.6 (step 7 in procedure compute_selectionMDD).

The final phase determines the actual bindings for each component. The bindings are generalized bindings with one case for each component design. Let us examine how the case label is determined for the component Head and the design P10_047. An MDD is formed by restricting the selection MDD to the cases where P10_047 is used for Head, yielding the MDD of Figure 6.7 (step 4 of procedure compute_bindings_with_options). Then, all non-terminal nodes associated with supplemental option variables are merged with the terminal true node and the MDD is reduced to the MDD in Figure 6.8 (step 5 of procedure compute_bindings_with_options). This MDD can be converted into the case label (Disp ={1835,2074}) * ($Ign ={'dual_elec','true_dual'}) (step 7 of procedure compute_bindings_with_options). The other case labels are determined similarly.

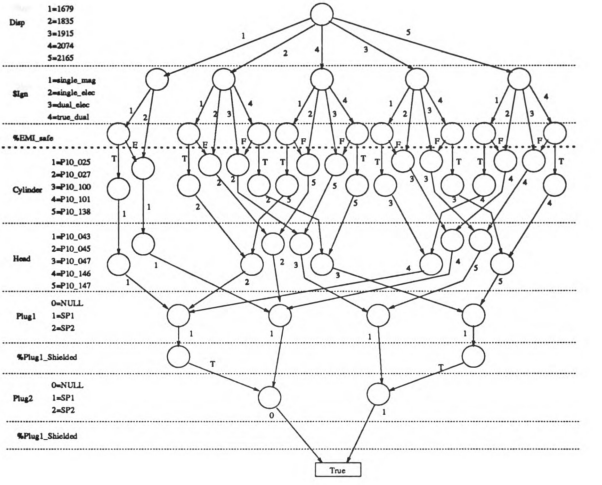


Figure 6.5. MDD for Constraints

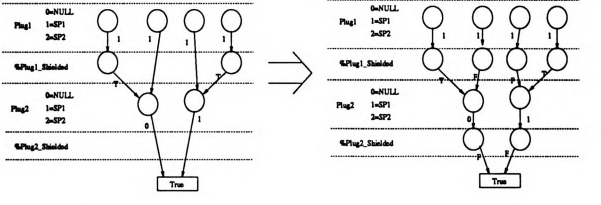


Figure 6.6. Removing ambiguity by adding non-terminal nodes

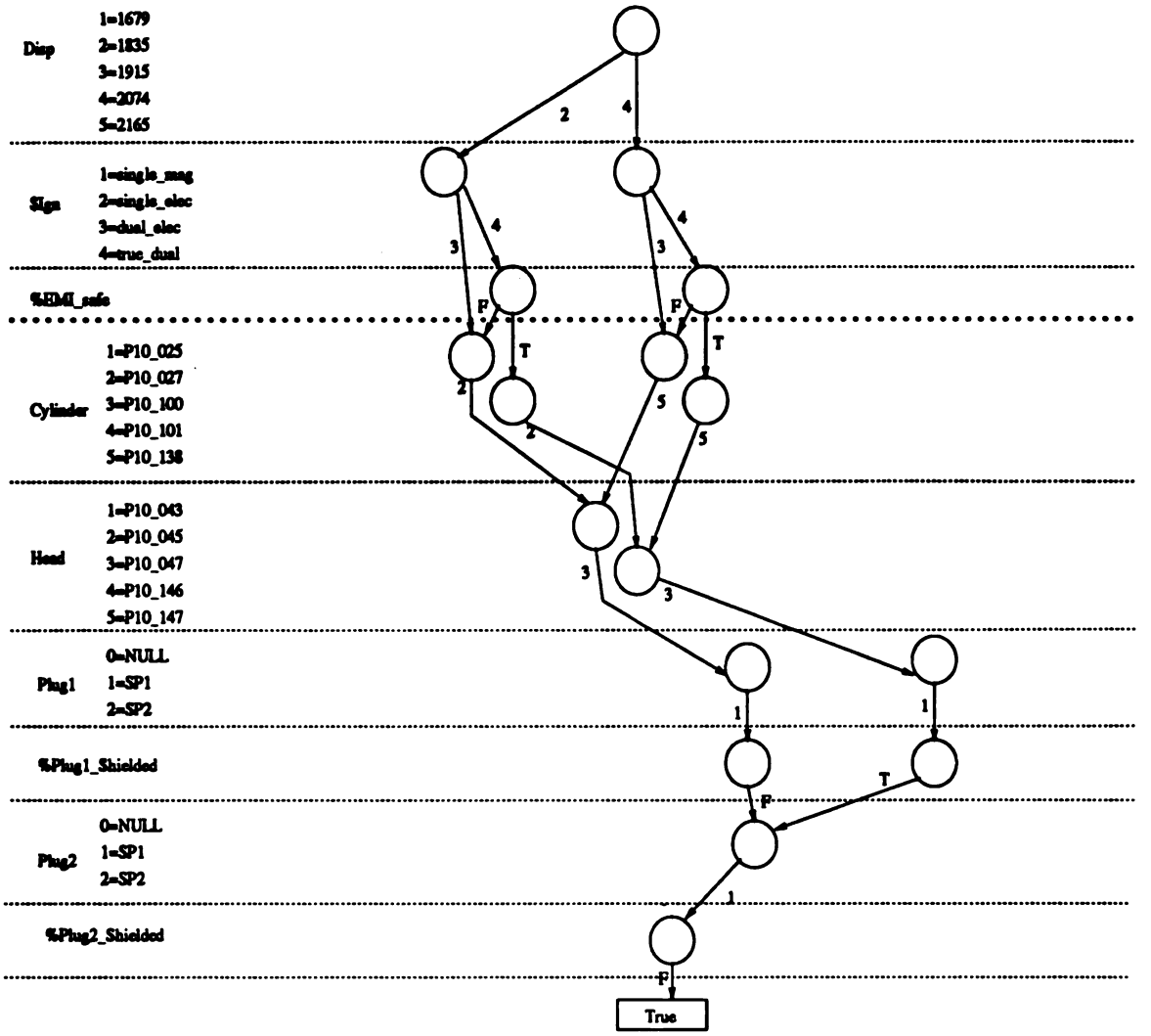


Figure 6.7. MDD for Cases where P10_047 is Used

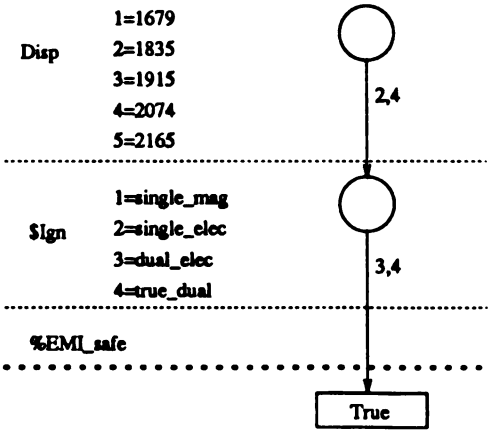


Figure 6.8. Primary Region MDD for Cases where P10_047 is Used

For the spark plugs, the designer must also find string expressions which indicate whether the %Shielded option should be selected or not. Let us look at the case when SP1 is used for Plug1. By restricting the selection MDD to the case where the supplemental option variable %Plug1_Shielded takes on the value TRUE, he gets the MDD of Figure 6.9 (step 11 of procedure compute_bindings_with_options). Then, all non-terminal nodes associated with supplemental option variables are merged with the terminal true node and the MDD is reduced to the MDD in Figure 6.10 (also in step 11). This MDD tells him to add the string expression `:(Disp = 1679) * ($Ign = 'single_mag') * %EMI_safe + (Disp >= 1835) * ($Ign = 'single_mag','true_dual') * %EMI_safe): -> (%Shielded)` to the dynamic binding for SP1 (step 12 of procedure compute_bindings_with_options). The expressions generated by the algorithm are often messier than what a person would produce. This expression can be simplified considerably.

In this example, the designer specified the components for 36 different variants and determined that 4 others were not possible. He examined 900 different combinations of component designs for each variant. However, many of the constraints were evaluated for a large number of different combinations without enumerating the combinations. Combinations which failed these simple constraints were not considered when the more complex constraints were evaluated. Therefore, only a very small fraction of the combinations had to be evaluated individually.

6.3 The recursive component selection problem

Several researchers have addressed another variation of the component selection problem called the recursive component selection problem. This variation is discussed

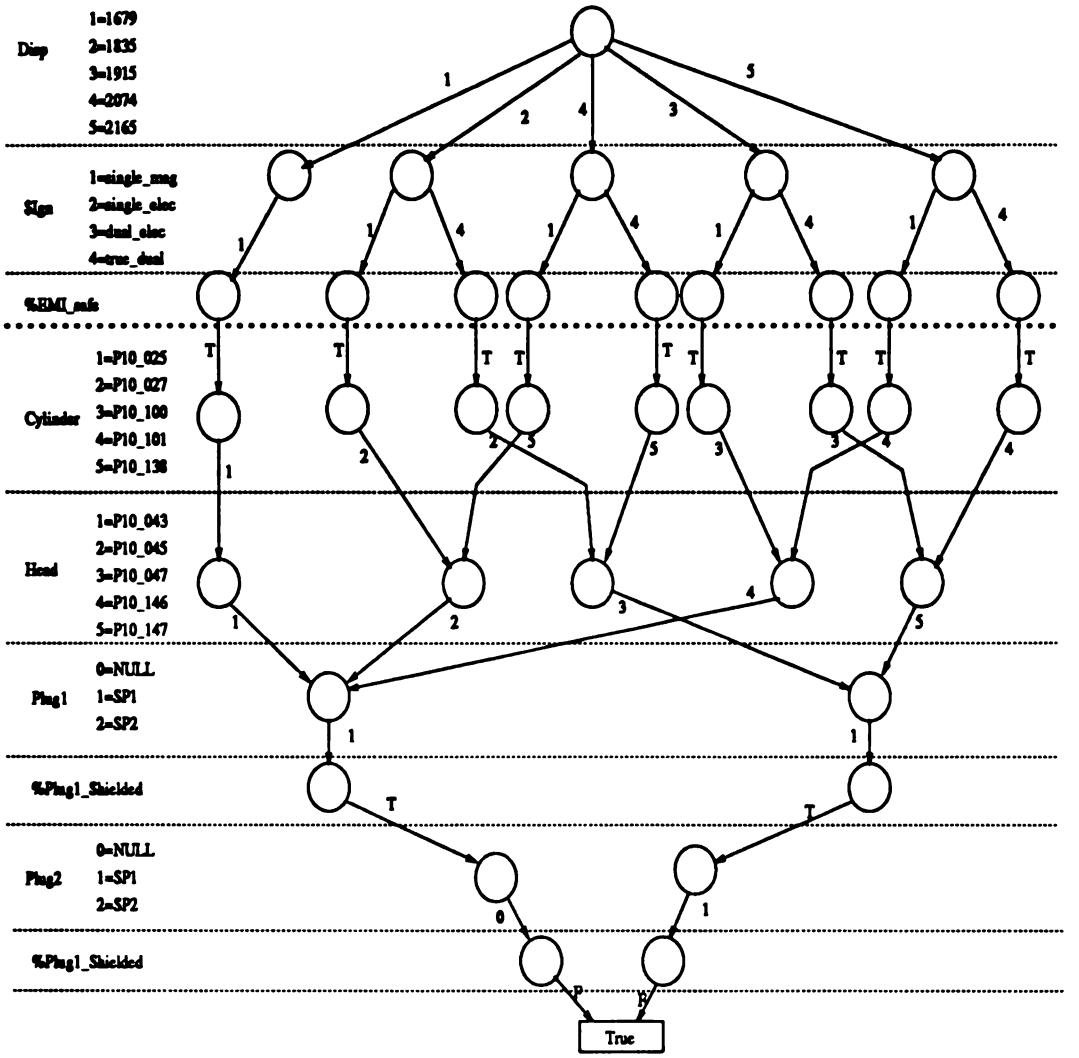


Figure 6.9. MDD for Cases where Shielded Spark Plug is Used

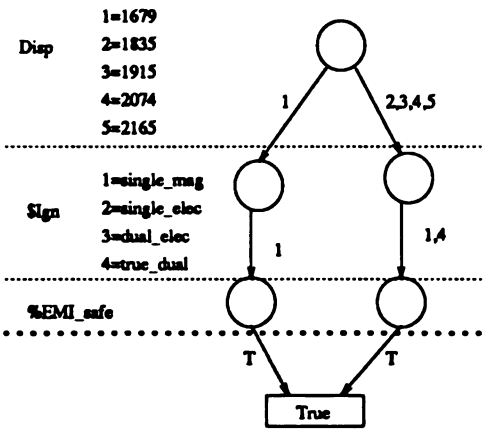


Figure 6.10. Primary Region MDD for Cases where Shielded Spark Plug is Used

briefly to clarify the relationship of the proposed algorithms to those of other researchers. No new algorithms are proposed for this variation. In the recursive component selection problem, the candidate component designs may themselves have unspecified components which must also be specified.

Kott et. al. [38] describe a system called the Configuration Tree Solver which automates the component selection process for "nearly decomposable" artifacts. Information about what structures are available is contained in an and-or tree called a *configuration knowledge tree*. The *or-nodes* represent different implementations of a design. The *and-nodes* represent decompositions of a design into components. A *configuration* is a subtree such that for any or-node in the configuration, exactly one child is in the configuration and for any and-node in the configuration, all children are in the configuration. The objective is to find a configuration that satisfies a set of constraints. The Configuration Tree Solver maintains a list of *partial configurations*, selects the most promising, and refines it by making a decision for an or-node. The new partial configuration is then evaluated according to how well it satisfies the constraints and the process is repeated. The process is facilitated by *local specialists* that contain domain specific knowledge. There are two types of local specialists, *or-specialists* and *and-specialists*, which are associated with or-nodes and and-nodes respectively. Or-specialists choose an alternative for a particular component subject to the constraints imposed by previous decisions. And-specialists decide which component should be chosen first, which is important because the first component chosen constrains later choices.

Brown and Chandrasekaran [40, 6, 41] developed a language called Design Specialists and Plans Language (DSPL) for describing a hierarchy of specialists which search the design space. Each *specialist* may have a variety of *plans* for completing its *task*. Plans consist of a series of *steps* which can be either simple steps or can invoke other specialists.

Kim [17] also addresses the recursive component selection problem using a separate algorithm than mentioned previously for combinational circuits. He takes advantage of the fact that some of the constraints apply to only one component. He uses these constraints first to reduce the number of candidates for each component. Then, he enumerates the combinations that remain and evaluates the remaining constraints for each combination individually. Since the number of combinations to be evaluated is very large, this phase dominates the solution time. If some of the candidate designs themselves have undetermined components, he applies his algorithm to the component designs first to enumerate all of the possibilities.

CHAPTER 7

Change Notification

Consider the following scenario. After completing the design of the head assembly and verifying that it satisfied the constraints, a designer started work on other projects. The designs of the components were all supposedly stable. However, a problem was discovered with the cylinder design and it was modified. These changes could influence the head assembly, so some of the constraints of the head assembly need to be re-checked. Unfortunately, it is difficult for the cylinder designer to anticipate the ramifications of his change, and the head designer has no reason to go back and re-check constraints unless he is made aware of the change. A mechanism is needed to notify appropriate individuals when design changes are made.

The notification mechanism must propagate change information to the appropriate people without inundating the designers with meaningless messages. Notifications are requested by posting *interest*. Designers post interest in designs that influence their own designs. When data is modified, the data management system checks for applicable interest and notifies the appropriate designers. Systems that provide notification differ in two primary respects:

1. what types of events trigger notifications, and

2. how designers are notified.

This chapter discusses how these issues may be handled within the proposed data model. First, however, some existing systems that offer change notification functionality are discussed briefly.

DVSS [18] automatically generates notification messages whenever conflicting accesses to data are detected. For example, if two users check out a version, modify it, and then check it in again, both users receive messages warning them that separate derivation paths have been created. Users may also explicitly define certain events that should result in notifications.

In [42], Chou and Kim discuss a system which notifies users when referenced objects are updated or deleted. The user specifies which references should be monitored. The system maintains two timestamps for each object: the time of the last update, and the time that changes to referenced objects were last approved. Whenever a referenced object has a update timestamp later than the last approved timestamp, a notification is generated. The notification can be a message to the owner of the referencing object or a flag which warns any user of the referencing object that changes have not been approved.

Agent systems [43] may be used for change notification. Instead of relying on the data management system to recognize that a change has occurred, the interested party runs an agent program that monitors the database. This implementation is most useful when many different data sources are involved, such as news feeds. When a CAD framework is managing the data, it is more natural to implement the notification functionality as a framework service.

7.1 Notification triggers

Whenever a designer posts an interest, he must specify what event will trigger a notification. In this section, the types of events that may be used are identified.

Modification of a Representation: The modification of a representation may imply, for example, that constraints need to be re-checked. Notification would be important if the designer making the modifications does not have primary responsibility for the overall design.

Creation of a Derived Representation: A derived representation may be created either to explore an improvement or to fix a bug in a frozen representation. In either case, the designer with responsibility for a design version using the representation may want to consider using the derived version instead.

Modification of a Design Version: When a design version is modified, either by modifying its representations or changing which representations it uses, designs which are influenced by it should be re-checked. Other designs would be influenced only if the design version is current for some workspace or is referenced via a static binding.

Violation of a Condition: A designer referencing this design might not care about all modifications, but rather only those that cause some condition to become violated. The condition would be specified using the same language as constraints and checked following each modification.

Change of Design Version Status: A designer may want to be notified when a design version he references is marked invalid.

Creation of a Derived Design Version: If a design references a design version with a static binding, the designer may want to consider using the updated version.

Release of a Design Version into a Workspace: If a design is referenced with a dynamic binding, then releasing a new design would influence the properties

of the referencing design. The designer of that design may wish to re-check constraints.

Change of Option Variables or Restrictions: If the option offerings are changed, the design may be instantiated by other designs with a different set of option values. The bindings may become impossible to evaluate (if they refer to choices that are no longer available) or ambiguous. The referencing designs must be re-checked.

Change of Constraints: If the constraints are changed, any design versions in use should be re-checked. If the changed constraints are environment constraints, referencing designs must re-check them. Also, although the framework's constraint checking mechanism does not utilize constraints on components, designers may consider them manually when verifying designs. If this has been done for designs that reference the changed design, those designs should be re-checked.

Creation of Child Design Object: When a new design is created, other designers may want to use the new design as a component in their own designs.

For example, the designer of the head assembly would want to post interests that would be triggered by the following events:

- Assuming that all components are referenced by dynamic bindings and that stable designs are placed in a workspace called "stable," the designer should be notified whenever any new design is released into the "stable" workspace for any of the component design objects.

- He should be notified whenever any of the design versions that are currently in the "stable" workspace for any component design objects are modified. This

trigger also covers any modification to representations used by those design versions.

- He should be notified whenever a referenced design version is changed from "valid" to "invalid" status.

- He should be notified whenever the option variables or option restrictions are changed for any components. If other people have authority to change the option variables and restrictions for the head assembly, he should also be notified when option variables or restrictions are changed.

- Similarly, if other people have authority to change the constraints for the head assembly, he should be notified of any changes to constraints.

- He should be notified when new design objects are created that are children of the non-leaf design objects `Cylinder`, `Cylinder_Head`, etc. Incorporating these new objects into his design may improve its performance or reduce its cost.

The modifications to the cylinder design would probably be done by creating a derived design version, editing it, and releasing it into the "stable" workspace after it was verified. The head assembly designer would be notified upon release.

7.2 Notification methods

Once a trigger event has been detected, there are many possible methods for notifying a designer. Sometimes, it is desirable to use a method that interrupts his work so he can respond immediately. At other times, it is sufficient to place a message somewhere where he will see it when he checks. Sometimes, it may also be necessary to prevent questionable data from being used until it is re-verified. In this section, methods that could be employed with the proposed data model are identified.

Message to Representation or Design Version: The message handling system of the data model may be employed by requesting a message to a particular representation or design version. A representation type could easily be implemented specifically for the purpose of receiving these messages. It would append them to a file for later review by the designer.

Change Status of Design Version: It may be best to mark a design version invalid so that others do not use it. A user flag determines whether bindings to invalid design versions succeed or fail.

Electronic Mail Message to User: The notified designer would probably check electronic mail much more frequently than he checks the notification representations described above. Therefore, notifications that require quick attention should use this method.

Re-checking Constraints: If the constraints can be checked quickly by the framework, the designer may want to be notified by the above methods only when the change results in a constraint violation.

Posting / Retraction of Interest: To avoid frequent interruptions, the designer may want to retract an interest after the first notification and replace it after he has taken appropriate action.

Unix Message or Signal to a Process: A process that displays information may post an interest so that it can update the display whenever the information is changed.

Part IV

Design Methodology Management

Previous chapters described constructs for managing the data generated during design and described framework services that can save designers time and reduce errors. However, further improvements in product quality and reductions in cost and time to market require improving the design process itself. In order to improve the design process, it must be possible to track the design process, and designers must be encouraged to use an approved process.

Traditionally, designers used whatever process seemed convenient to them at the time. It was virtually impossible to determine how a past design was produced. Moreover, this practice often resulted in steps being skipped. Designs were not checked for important criteria such as manufacturability until it was too late to make changes.

Design methodology management is the selection and execution of an appropriate sequence of tools to produce a design description from available specifications. Selecting a sequence of tools can be a daunting task due to incompatible assumptions and data formats among tools. Also, to support a higher degree of automation, CAD frameworks must be able to select and execute tools automatically for frequently repeated tasks, enabling designers to concentrate on higher level decisions.

Methodology management systems can be characterized by two criteria:

Methodology Specification How are legal methodology choices specified? Ideally, the choices should be specified in a form that is maintainable as new tools are acquired and new methodologies are developed.

Execution Environment How is a particular methodology chosen at run time? Designers must be informed of their choices and given guidance about which choices are preferred.

The proposed framework includes extensive methodology management functionality. In Chapter 8, the proposed specification formalism, called *design process grammars* is discussed. In Chapter 9, an execution environment which helps the designer choose between alternative methodologies is proposed.

The primary advantages of the proposed methodology management system are:

Formalism The system is based on a strong theoretical foundation, enabling administrators to analyze how the system will operate with different methodologies.

Parallelism The system allows several alternatives to be explored simultaneously, enabling designers to make better use of idle computing resources.

Extensibility New tools can be integrated into the system easily by adding productions and control agents.

Flexibility Many different control strategies can be used. Several control strategies can even be mixed within the same design exercise.

CHAPTER 8

Methodology Specification

An important ingredient in methodology management is a formal representation of design processes. Many different formalisms have been proposed.

Directed graphs are used to represent design processes in systems such as the program evaluation review technique (PERT) and critical path method (CPM). In these representations, the nodes represent milestones and the arcs represent tasks. These representations were invented to help managers schedule resources when the duration of each task is known, at least probablistically. However, the duration of design tasks is very difficult to predict, so these methods have limited use in design.

Steward [44] proposes a system for describing design processes using a precedence matrix. A process consisting of n tasks is represented by a matrix with one row and column per task. The rows and columns are labeled in the same order. An entry in row A, column B implies that task B uses data generated by task A. The objective is to re-order the tasks such that the matrix is lower triangular. However, circular information flows make this impossible implying that some tasks must be done iteratively. The data represented by entries above the diagonal must be guessed on the first iteration and then refined in later iterations. Gebala et. al. [45, 46] extend Stewards representation to numerical degrees of dependence as opposed to Boolean entries. These numerical values provide assistance is determining which data to guess

on the initial iteration. These matrix methods are suitable only when the set of tasks is constant.

Systems such as Configuration Trees [38] and DSPL [6, 40, 41] can also be treated as methodology specifications. Configuration Trees organize the hierarchy of specialists according to the component hierarchy of the artifact being designed. DSPL organizes the hierarchy according to the decompositions of the design tasks. In some domains, but not all, the task decomposition is identical to the component hierarchy. Organizing the hierarchy according to task decomposition is a more general approach.

Nelsis [10] provides a graph formalism, called *flowmaps*, for representing sets of methodologies. An example flowmap is shown in Figure 8.1. Each task is referred to as a *functional unit.* Flowmaps have a hierarchical structure in which some functional units correspond to *activities* (atomic operations) and others correspond to more detailed flowmaps. Some flowmaps are flagged to indicate that they should be executed automatically as soon as the input data is available. Functional units are connected to each other by *channels*, which have specific *datatypes.* The *input (output) ports* of a functional unit indicate what channels the functional unit receives (supplies) data on (to). Some of the input ports are *optional* (indicated by a solid circle), indicating that the functional unit can run with or without that data. These optional ports are very important for iterative design processes, which result in cycles in a flowmap. There is a distinction between output ports based on whether the data is added to the input design object, called *extension*, or placed in a separate design object, called *modification.* When tasks are repeated, previous extension port outputs and data derived from them must be invalidated. New versions are created for data on extension ports. If two functional units have output ports to the same channels, they are considered to be alternatives. Users must statically specify a preference among alternatives or choose among them at run time. If two or more functional units do not have any outputs, flowmaps do not indicate whether they are alternatives or must

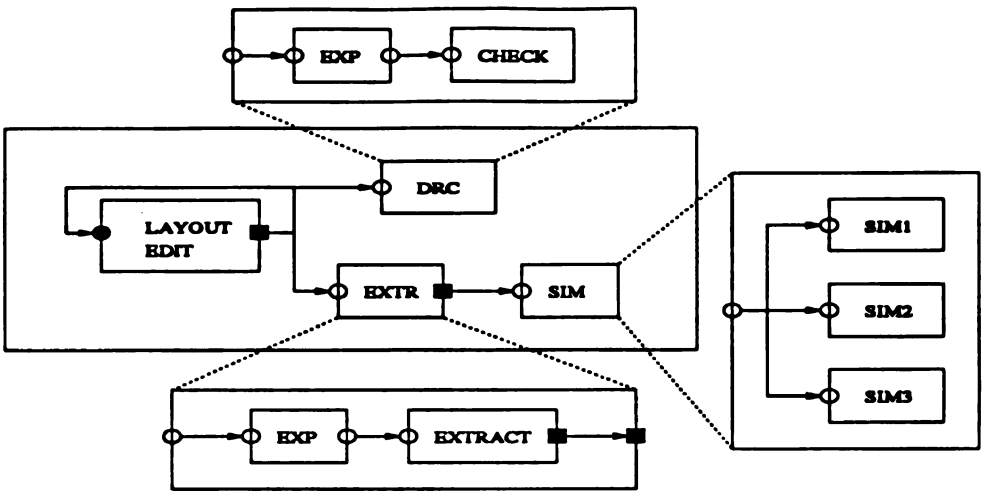


Figure 8.1. A Nelsis Flowmap (from [10])

all be completed (see SIM1, SIM2, and SIM3 in Figure 8.1). Further, Nelsis does not provide any mechanism to pursue multiple versions simultaneously.

Sutton et. al. [47] use a similar graph formalism called a *task graph*. Tools and data are both treated as *design entities*. Each design entity may have at most one functional dependence indicating the tool that was used to create it and any number of data dependences indicating the input data to that tool. The set of legal task graphs is determined by a *task schema* as shown in Figure 8.2. The task schema indicates which tools (f arrows) and data types (d arrows) can be used to create each type of data. When more than one method is allowed, subtypes are used, as illustrated by the circuit and layout types. As designers work, they build task graphs in a bottom-up manner, as long as the nodes that are added to the task graph have the necessary dependencies as indicated in the task schema. More than one alternative may be pursued in parallel, although there is a risk of using incompatible data in later steps.

In this chapter, a formalism called *process flow graphs*, which represents individual methodologies, is introduced. A type of graph grammar called a ***design process grammar*** is used to document what methodologies are available in a framework. Designers

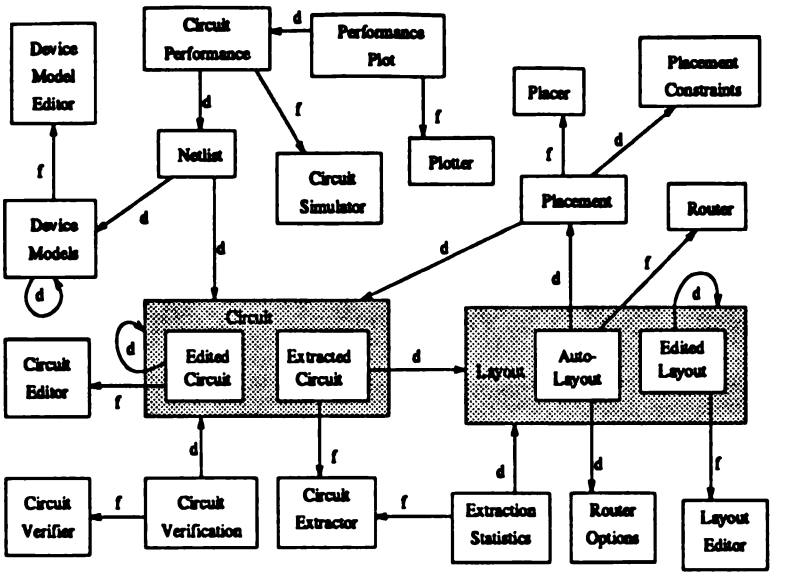


Figure 8.2. A Task Schema (from [47])

build process flow graphs in a top-down manner by applying productions from the design process grammar to expand nodes that represent abstract tasks. When tasks are repeated, such as in iterative design, there are multiple versions of its outputs. A formalism called a *versioned flow graph* is used to represent design processes with repeated tasks. Conditions for versions of specifications to be *compatible* are also identified. Alternative methodologies are very explicit, making it easier to indicate which methodologies should be chosen. Also, design process grammars are context sensitive, making restrictions due to incompatible data formats explicit. By analyzing the grammar, framework administrators can either prove that a tool set is capable of performing all of the required tasks or can identify missing capabilities.

The proposed formalisms are more natural than the flowmaps of Nelsis. When using a Nelsis flowmap, a design has to examine the output relationships between tasks in order to determine if they are alternatives or must both be done. Alternatives are very clear in the proposed formalisms. Once a particular alternative is chosen, the

other alternatives are not included in the process flow graph where they could confuse a designer.

8.1 Process flow graphs

Process flow graphs describe the information flow of a design process. Formally, a process flow graph is a bipartite acyclic directed graph of the form $G = (T, S, E)$, where

T is the set of task nodes. Each task node is labeled with a task description. (Task nodes are drawn as circles.)

S is the set of specification nodes. Each specification node is labeled with a design object name and a representation type. (Specification nodes are drawn as rectangles.)

E is the set of edges indicating which specifications are used and produced by each task. Each specification must have at most one incoming edge. Specifications with no incoming edges are assumed to be inputs of the design exercise.

$T(G), S(G), E(G)$ are the sets of task nodes, specification nodes, and edges of graph G, respectively.

Figure 8.3 shows a process flow graph that describes a possible design exercise undertaken by the aircraft engine company to assist a customer.

As discussed in Section 2.2, the various representation types form a class hierarchy, where each child is a specialization of the parent. There may be several incompatible children. For example, `AutoCAD file` is a child of `CAD Data`, but is not interchangeable with `CADAM file`. Descendant representation types are also called subtypes. Representation types of specification nodes are used to avoid data format incompatibilities between tools.

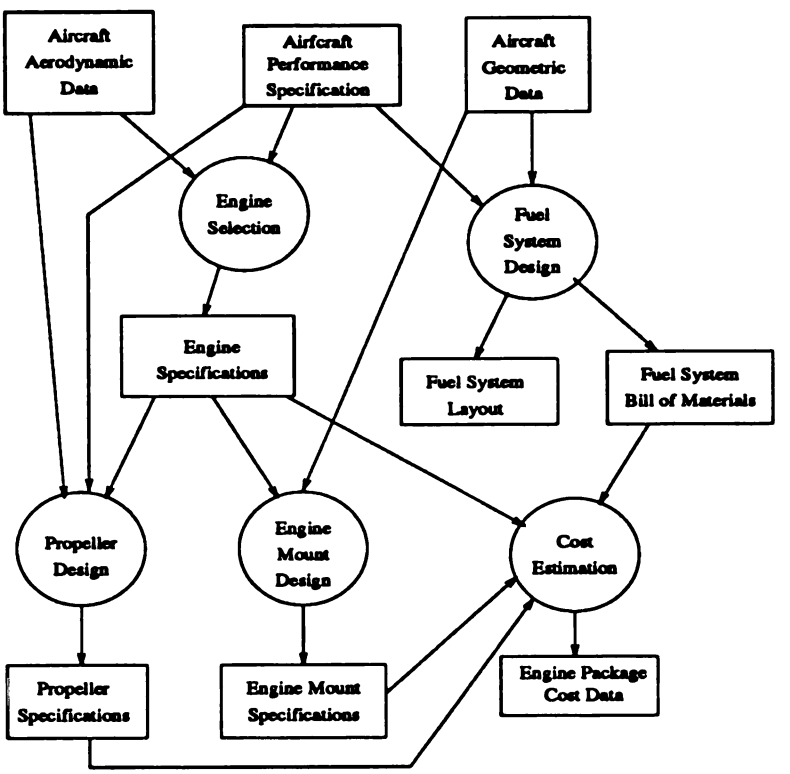


Figure 8.3. A Sample Process Flow Graph

Task nodes can be either terminal or non-terminal. A terminal task node represents a run of an application program, which is commonly called a *tool invocation.* The representation type of the outputs of a terminal task node must have no subtypes. Terminal task nodes are drawn with double circles. Non-terminal task nodes represent abstract tasks, which could potentially be done with several different tools or combinations of tools. Process flow graphs can describe design processes to varying levels of detail. A graph containing many non-terminal nodes indicates roughly what should be done and what information is desired without describing exactly which tools should be used. Conversely, a graph in which all nodes are terminal, called a *terminal graph*, completely describes a design process.

The following definitions are used:

$In(N)$ is the set of input nodes of node N: $In(N) = \{M \in T \cup S | (M, N) \in E\}$.

$Out(N)$ is the set of output nodes of node N: $Out(N) = \{M \in T \cup S | (N, M) \in E\}$.

$I(G)$ is the set of input specifications of graph G: $I(G) = \{N \in S(G) | In(N) = \emptyset\}$.

The representation types of nodes in $I(G)$ must have no subtypes.

8.2 Design process grammars

Graph grammars provide a convenient means for transforming process flow graphs into progressively more detailed process flow graphs. The user specifies the overall objectives by supplying the initial graph, which indicates what input specifications are available, what output specifications are desired, and what abstract tasks should be performed. The start graph is progressively modified using a graph grammar, called a *design process grammar.* The non-terminal task nodes, which represent abstract tasks, are replaced by subgraphs of less abstract tasks and intermediate specifications. The

output specification nodes are also replaced by nodes that may have a more specific format (a child representation type).

The productions in a graph grammar permit the replacement of one subgraph by another. A production in a design process grammar can be expressed as a tuple $P = (G_{LHS}, G_{RHS}, \sigma_{in}, \sigma_{out})$ where

G_{LHS}, G_{RHS} are process flow graphs for the left side and right side of the production, respectively, such that $T(G_{LHS})$ is a single, non-terminal task node representing the abstract task to be replaced.

σ_{in} is a mapping from $I(G_{RHS})$ onto $I(G_{LHS})$ indicating the correspondence between input specifications. Types must match exactly.

σ_{out} is a mapping from $S(G_{LHS}) - I(G_{LHS})$ to $S(G_{RHS})$ indicating the correspondence between output specifications. Each output specification must map to a specification with the same type or a subtype.

Figures 8.4 and 8.5 illustrates some productions for the tasks `Fuel System Design` and `Propeller Design`. The mappings are indicated by the numbers beside the specification nodes. The vertical bar is a shorthand notation to indicate multiple rules with the same G_{LHS} but different G_{RHS}'s. Alternative productions may be necessary to handle different formats (as in Figure 8.4), or because the right hand sides perform differently in different situations (as in Figure 8.5).

Let A be the non-terminal task node in $T(G_{LHS})$ of production P and A' be a non-terminal task node in the original process flow graph, G. Formally, P matches A' if:

1. A' has the same task label as A,

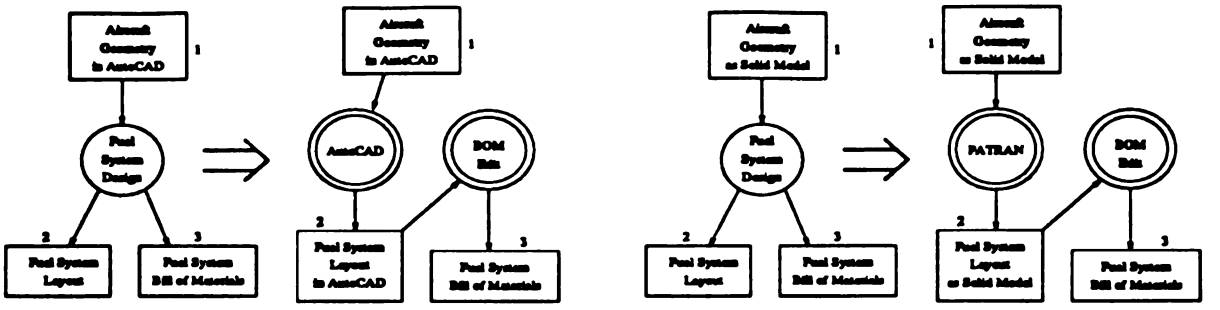


Figure 8.4. Alternative productions based on input format

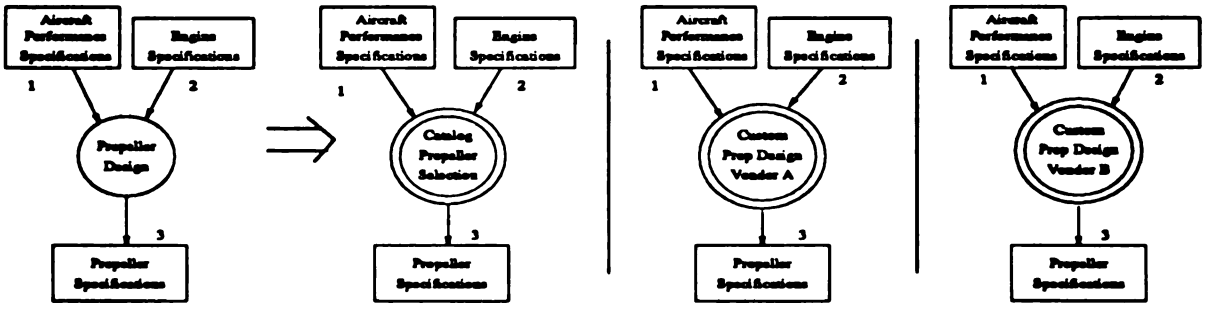


Figure 8.5. Productions indicating alternative algorithms

2. There is a mapping, ρ_{in}, from $In(A)$ onto $In(A')$, indicating how the inputs should be mapped. For all nodes $N \in In(A)$, $\rho_{in}(N)$ should have the same type as N or a subtype.

3. There is a mapping, ρ_{out}, from $Out(A')$ onto $Out(A)$, indicating how the outputs should be mapped. For all nodes $N \in Out(A')$, $\rho_{out}(N)$ should have the same type as N or a subtype.

The mappings are used to determine how edges that had connected the replaced subgraph to the remainder should be redirected to nodes in the new subgraph.

Once a match is found in graph G, the production is applied as follows:

1. Insert $G_{RHS} - I(G_{RHS})$ into G. The inputs of the replaced task are not replaced.

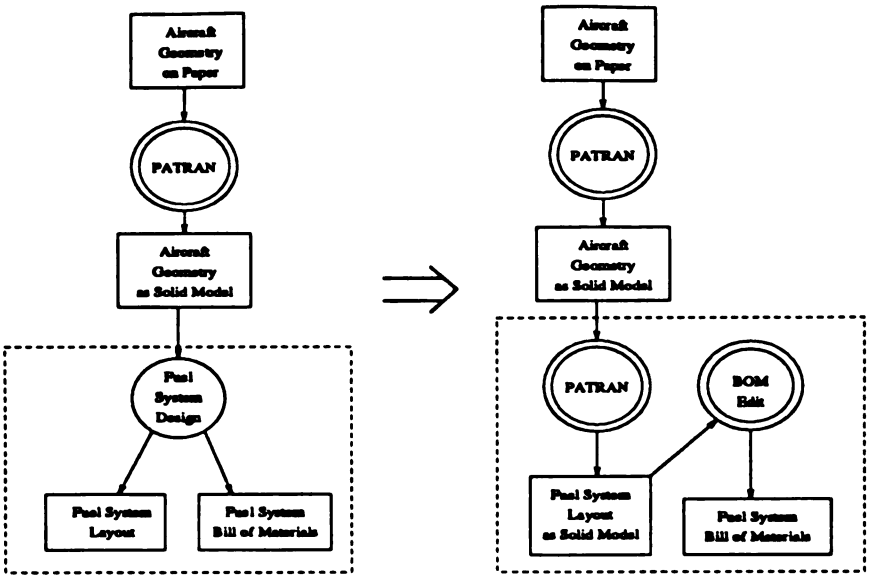


Figure 8.6. A Sample Graph Derivation

2. For every N in $I(G_{RHS})$ and edge (N, M) in G_{RHS}, add edge $(\rho_{in}(\sigma_{in}(N)), M)$ to G. That is, connect the inputs of A' to the new task nodes that use them.

3. For every N in $Out(A')$ and edge (N, M) in G, replace edge (N, M) with edge $(\sigma_{out}(\rho_{out}(N)), M)$ to G. That is, connect the new output nodes to the downstream tasks which use them.

4. Remove A' and $Out(A')$ from G, along with all edges incident on them.

Figure 8.6 illustrates a derivation using a production from Figure 8.4. The dotted lines outline the subgraph that is replaced.

How does the data generated during the execution of a methodology fit into the proposed data model? When Cockpit processes the start graph at the beginning of a design exercise, it creates a new design object for each different design object name that appears in a specification node. If an existing design object has that name, the newly created object is a child of that design, otherwise it has no parent design in the classification hierarchy. Cockpit must create a new name for the new design object

by adding some characters to the name provided). Each of these new design objects starts with a single design version, which contains slots for each specification node with that design object name. The slot for each input specification node points to the corresponding representation. The slots for output nodes initially have null pointers.

When a production is applied, new intermediate specification nodes (nodes that are not inputs or outputs of the task being planned) may be introduced. If these new nodes have a design objects name different than the name of any of the input and output nodes in the production, a new design object and design version is created for them. Otherwise, a new slot is added to an existing design version.

When a tool is executed, the pointers for the slots corresponding to its outputs are set to the newly created representations. The new representations are "derived from" the inputs to the task. Section 8.4 discusses how additional design versions are created to explore alternatives.

8.3 Guaranteeing success

In this section, *completeness* of grammar symbols is discussed. Completeness guarantees that a process flow graph with no non-terminal task nodes can be generated from an initial graph. Without this guarantee, it is dangerous to start execution of any of the tasks before completely generating the process flow graph. The designer might reach a dead end where, after investing considerable effort, there are no tools to complete the job from the present state. Being able to start execution before planning is completed is important because information generated by executing some tasks can be very useful in planning others.

A task label is *complete* with respect to certain input and output types if it is possible to produce acceptable output types from any combination of possible input types for that task. Formally, let $I = \{in_1, in_2, ..., in_n\}$ and $O = \{out_1, out_2, ..., out_m\}$

be lists of types for a task's input and output specifications, respectively. Let $I' = \{in'_1, in'_2, ..., in'_n\}$ be a list of types such that each in'_i is in_i or a subtype of in_i and in'_i has no subtypes for $1 <= i <= n$.

Definition 8.1 *A terminal task label T is complete with respect to input types I and output types O if and only if: for every possible I', there exists an* $O' = \{out'_1, out'_2, ..., out'_m\}$ *such that*

i. $out'_i = out_i$ *or a subtype of* out_i *for* $1 <= i <= m$, *and*

ii. the tool represented by T can take inputs with types I' and produce outputs with types O'.

Definition 8.2 *A non-terminal task label T is complete with respect to input types I and output types O if and only if: for every possible I', productions exist to transform a non-terminal task node with label T, inputs with types I', and outputs with types O into a terminal graph in which all task nodes are complete with respect to their inputs and outputs.*

Completeness of the nodes in the initial graph guarantees success of design planning. If a task node, N, has input types I, those types may be changed to types I' by productions applied to predecessor task nodes (tasks which supply data to N). However, if the label of N is complete with respect to I and its output types O, then it can operate on any I' that might occur. Notice that the output types O are never changed except by applying a production to N. If all task nodes in the initial graph are complete with respect to the specification types with which they appear, then there are tools available to transform the input specifications into outputs of the desired type. Users should avoid applying a production which includes a task node in G_{RHS} which is not complete with respect to the input and output types with which it appears.

Fortunately, a set of productions can be checked for completeness of non-terminal tasks using Algorithm check_completeness. A list is maintained of combinations of task label, input types, and output types such that the task label is complete with respect to the input and output types. The user must initiate the process by indicating the capabilities of the individual tools (listing the complete combinations for terminal task labels). New combinations are added by finding productions that match non-terminal task labels with certain input and output types such that all of the nodes on the right hand side are complete. The algorithm iterates until no new combinations may be added.

Before the algorithm is described in detail, the conditions for combining two or more complete combinations to form a new complete combination must be elaborated. Let all subtypes of specification type t be denoted by t_1 through t_k. If task label T is complete with respect to $I_i = \{in_1, ..., t_i, ..., in_n\}$ and $O_i = \{out_{1i}, ..., out_{mi}\}$ for $1 <= i <= k$, then it is also complete with respect to $I = \{in_1, ..., t, ..., in_n\}$ and $O = \{out_1, ..., out_m\}$ such that each out_{ji} is out_j or a subtype. This assertion follows directly from the definition of completeness for non-terminal task labels.

```
Algorithm check_completeness

Input:
  A set of productions
  A list, L_in, of complete terminal symbols,
    indicating what output types, O', may be produced
    from what input types, I', by each tool

Output:
  A list, L_out, of complete non-terminal symbols,
    indicating what output types, O', may be produced
    from what input types, I', for each abstract task

1. Initially, L_out is empty

2. For each production {

3.   Let S' be a mapping from specification nodes to types such
```

```
        that each node maps to the type of its label or a subtype.
        For each possible S' {

4.        If every task node on the right hand side appears in
          either L_in or L_out as complete with respect to the
          types its inputs and outputs map to {

5.          Add the task node on the left hand side of the
            production to L_out with the types its inputs map to
            and the types of its outputs (not mapped)

          }
        }
      }

6. Add to L_out any new complete combinations possible by
   combining combinations in L_out.

7. If any new productions were added to L_out, go to 2.
```

The following theorems state that Algorithm check_completeness finds all of the combinations that are complete and none that are not complete.

Theorem 8.1 *If the task label T with input types I and output types O is listed by Algorithm check_completeness, then it is complete with respect to I and O.*

Proof: Consider the ith combination added. The proof is an induction on i. The first combination added must be added by step 5 and the production must contain only terminal nodes on the right hand side. The combination is complete since applying the production would transform task node with label T, inputs with types I and outputs with types O into a terminal graph and all of the task nodes added are listed as complete in *L_in*, which is assumed to be correct. Therefore, the theorem holds for $i = 1$. The ith combination is added by either step 5 or step 6. If it is added by step 5, it is complete because the production could be applied to any node with label T, inputs with types I', and outputs with types O and all of the tasks nodes added by applying the production appear in either *L_in* or *L_out*, and

can therefore be transformed into terminal graphs by the induction hypothesis. If the combination is added by step 6, then any I' must already be covered by a combination in *L_out* and is therefore complete by the induction hypothesis. □

Theorem 8.2 *If a non-terminal task label, T, is complete with respect to input and output types I and O, then the combination is listed by Algorithm check_completeness.*

Proof: Consider a graph which consist only of a non-terminal task with label T, input specification nodes with labels I' and output specification nodes with labels O. If T is complete with respect to I' and O, then this graph can be transformed into a terminal graph by a sequence of i production applications. The proof is an induction on i. If $i = 1$, then there is a production which matches T with inputs I' and outputs O and has no non-terminal task nodes on the right hand side. If each of the terminal task labels on the right hand side is listed as complete with respect to its input and output types in *L_in*, then step 5 adds the combination T, I', O to *L_out*. Now, assume that any complete combination that can be transformed into a terminal graph by i or fewer combinations is listed in *L_out*. In the following iteration, Algorithm check_completeness finds that the first production in the sequence matches T with inputs I' and outputs O. By the induction hypothesis, all of the task nodes on the right hand side are listed as complete with respect to their inputs and outputs, so the combination T, I', O is added to *L_out* by step 5. Since the combinations T, I', O is added for every I', step 6 adds the combination T, I, O to *L_out*. □

8.4 Handling multiple versions

The previous section described how process flow graphs are generated by replacing abstract tasks with graphs of less abstract tasks. However, design involves a search

through the space of possible alternatives, implying that some of the tasks may be executed several times. For example:

- The first execution might not produce an acceptable result, so backtracking occurs and some of the decisions made on the first execution are changed.
- New information may become available which changes some of the decisions made on the first execution, such as approximate characteristics of the final design. In iterative design processes, this new information is a direct result of earlier executions and is refined in later executions.

Each time a task is repeated, it produces new versions of its outputs. Multiple executions of an abstract task often involve using a different production from the design process grammar, implying that the space of possible methodologies is being searched.

An extension of the process flow graph, called a *versioned flow graph*, captures this dynamic nature of design processes. Like a process flow graph, a versioned flow graph is a bipartite acyclic directed graph of the form $G = (T, S, E)$ with the same definitions for T, S, and E. However, the rules for applying a production are changed slightly. When a production is applied in a versioned flow graph, the task node being expanded, A', and its outputs are not removed. A production can be applied to A' again indicating that the task is to be repeated. Each time a production is fired, new specification nodes are generated for the outputs of the abstract task represented by A'. These nodes represent alternative versions of those specifications. The new task nodes are called *subtasks* of A', even if there is only one. Figure 8.7 shows how the derivation of Figure 8.6 would be carried out in a versioned flow graph.

In versioned flow graphs, specification nodes resulting from different assumptions can coexist, as shown in Figure 8.8. The bill of materials from alternative #1 must not be used in conjunction with the layout from alternative #2 in a later task, such

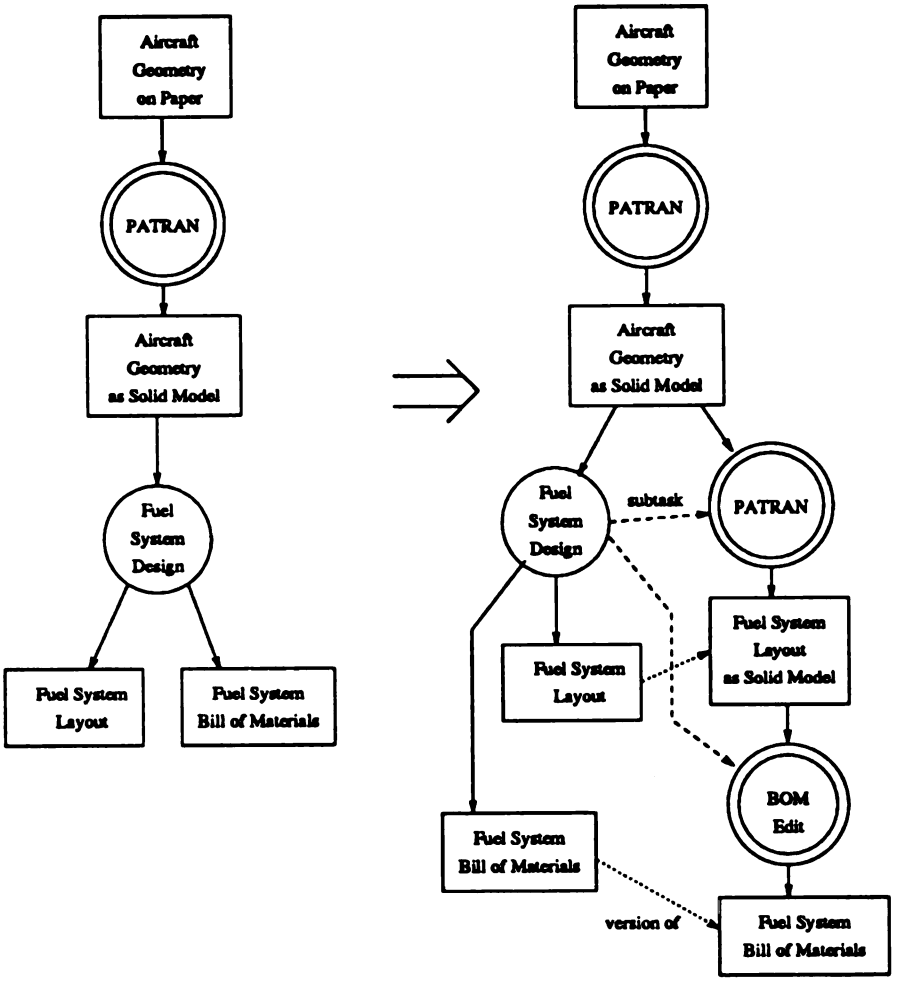


Figure 8.7. Sample Derivation in Versioned Flow Graph

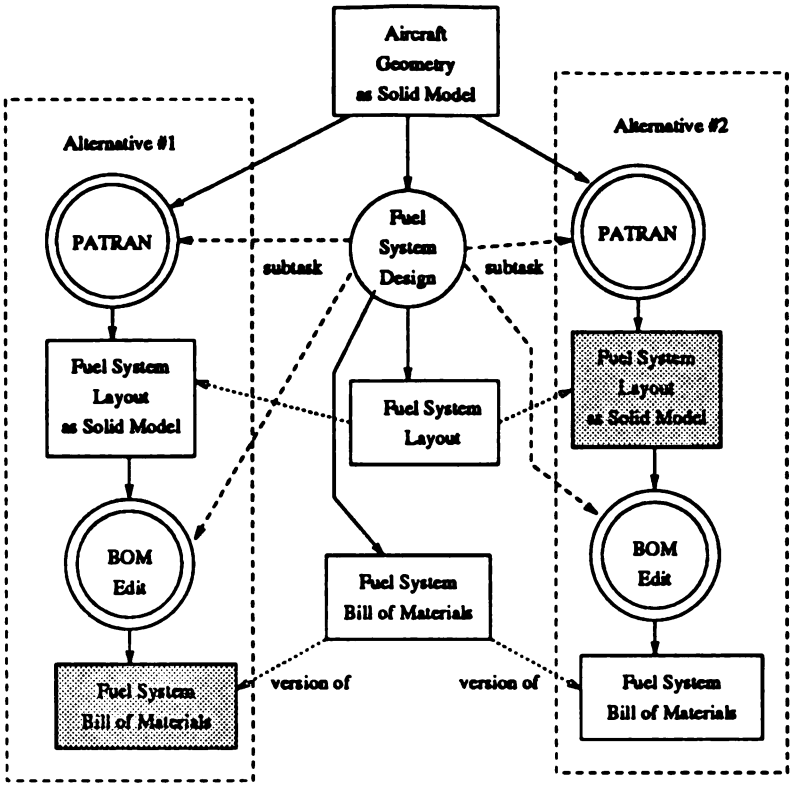


Figure 8.8. Incompatible Specification Nodes

as producing a shop order. Notice that these two nodes would never appear in the same non-versioned process flow graph because non-versioned process flow graphs do not allow alternatives. Prior to applying a production in a versioned flow graph, the non-terminal task node, input specifications, and output specifications must be checked for *compatibility.*

Definition 8.3 *Two or more nodes are called* compatible *if and only if a non-versioned process flow graph could be constructed that contains all of them.*

The above definition is not very practical for efficiently determining whether certain nodes are compatible. To develop an efficient algorithm for determining compatibility, it is necessary to define the sequence of production firings for a node. A

production firing can be characterized by the non-terminal task replaced, the production used, and the input and output mappings. The sequence of firings for a node is defined recursively as the firing in which the node was added to the graph concatenated to the sequence of firings for the task node replaced by that firing.

Theorem 8.3 *Two nodes are NOT compatible if and only if*

i. their sequences of firings contain different firings for the same non-terminal task, or

ii. the sequence of firings for one node includes a firing applied to the other node (if it is a task) or its source (if it is a specification).

Proof: If each sequence contains a different firing applied to non-terminal task N, then adding the first node to a non-versioned process flow graph would delete N, making it impossible to add the other node. If a production is applied to a task node N with output specification node S, then both N and S would be deleted from the non-versioned process flow graph. Any node with that firing in its sequence would be incompatible with N and S. If neither of the above conditions hold, then a non-versioned flow graph can be constructed by applying the sequence of firings for the first node and then applying any firings in the sequence for the second that have not already been applied. □

The set of compatible nodes produced by a sequence of firings is called a ***design state.*** In order to apply a production, all of the nodes involved must be included in the same design state. Applying the production removes the task node and its outputs from the design state, but not from the versioned flow graph. To pursue an additional alternative, a new design state is created. Productions fired in the new design state have no effect on other design states and vice versa.

Using the data model of Chapter 2, design states correspond to new design versions and workspaces. To create a new design state, a new workspace is created. Then, a new design version is created for each design object involved and the new workspace is set to map the design objects to the new design versions. The new design versions is initially identical to the previous design versions and has a "derived from" relationship with them. However, representations added to the new design version as tools complete are not added to the originals. Backtracking to the previous design state simply requires making the previous workspace current again.

CHAPTER 9

Execution Environment

Once a formalism is available for representing methodologies, frameworks can provide support to the user in selecting an appropriate methodology and executing it. Previous systems provide various degrees of assistance.

The simplest form of assistance is monitoring the designers' actions. In [51], Di Janni describes a *monitor* for CAD tools which models fixed methodologies using extended Petri nets. The VOV system [52] records the sequence as the designer executes tools. When an input file is modified, these systems help the user keep data consistent by invalidating output files or by repeating previous tool executions.

There are several systems which automatically determine what tools to execute. The Design Planning Engine of the ADAM system [53, 54] produces a ***plan graph*** using a forward chaining approach. Acceptable methodologies are specified by listing pre-conditions and post-conditions for each tool in a Lisp-like language. Estimation programs are used to guide the chaining. Ulysses [55] and Cadweld [56] are blackboard systems used to control design processes. A knowledge source, which encapsulates each tool, views the information on the blackboard and determines when the tool would be appropriate. Minerva [57] and the OCT task manager [58] use hierarchical strategies for planning the design process. Hierarchical planning strategies take

advantage of knowledge about how to perform abstract tasks which involve several subtasks.

In [16], Katz and Chang discuss validation scripts for *active equivalence constraints.* When an object with an active equivalence constraint is checked into a workspace, the validation script is executed to create new objects that are consistent.

In this chapter, the proposed execution environment is presented. Section 9.1 is a high level overview of the environments architecture. An example is used to illustrate its operation in Section 9.2. Finally, in Section 9.3, some of the implementation options of manager programs are discussed.

9.1 Execution environment overview

The architecture of the proposed framework is illustrated in Figure 9.1. The designer interacts with a program called Cockpit, which keeps track of the current status and informs the user of possible actions. To assist the user in choosing an appropriate action, Cockpit interacts with several manager programs, which encapsulate design knowledge. The manager programs provide ratings for the productions, invoke tools, and may perform some design tasks automatically. Manager programs must be maintained by tool integrators to reflect site specific information such as company design practices and different ways of installing tools.

Cockpit keeps track of the current state of the design process. Unlike the manager programs, Cockpit contains no task specific knowledge. Its information about the design process comes entirely from an input file indicating the set of possible tasks and what productions should be considered for each non-terminal task. For each task, and for each production, the input file indicates a manager program that encapsulates knowledge about that task or production.

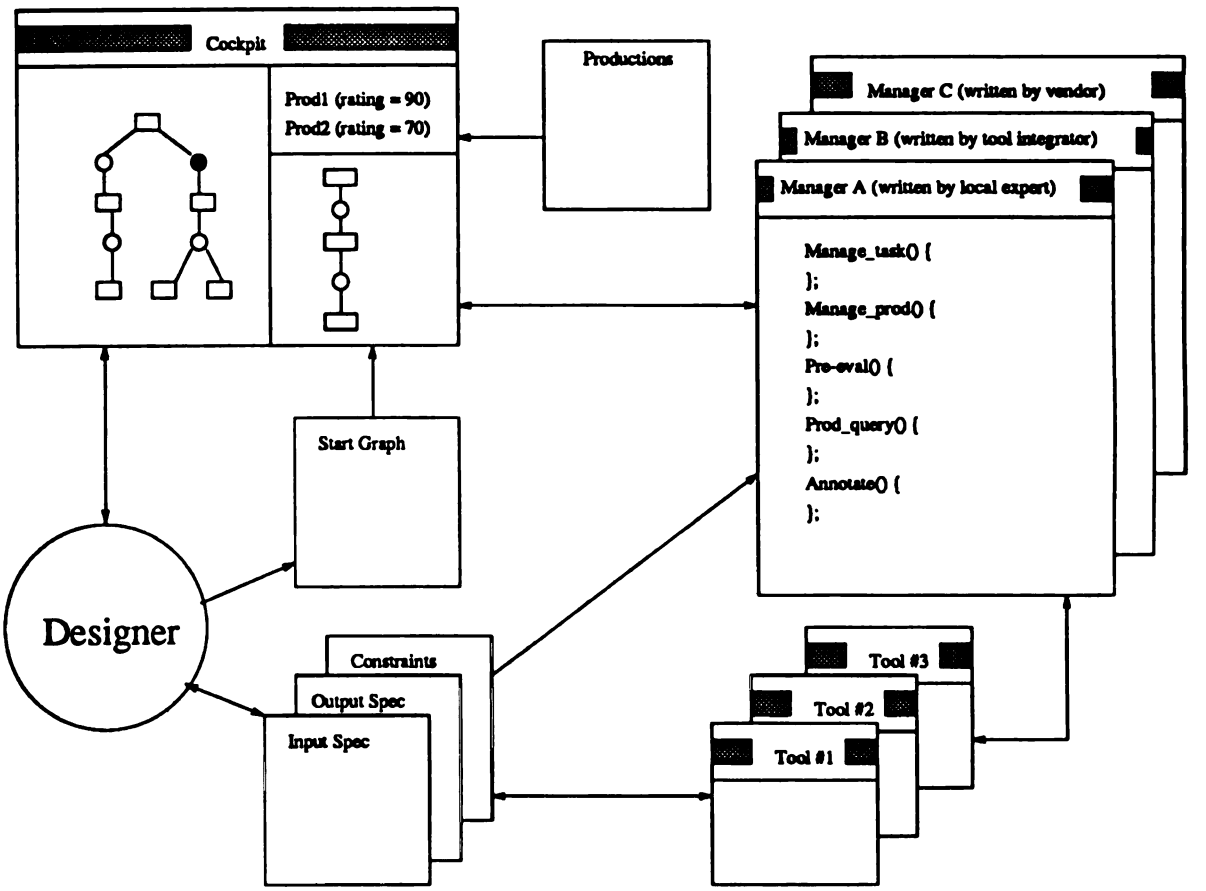


Figure 9.1. System Organization

The designer directs the design process by interacting with Cockpit. Cockpit displays a process flow graph indicating the current plan for completing the design exercise. Cockpit determines what productions are available for remaining non-terminal tasks. For each production, it sends a message to the corresponding manager program requesting that the manager compute a rating of the production's usefulness in the current situation. This information is displayed to the user. Initially, Cockpit waits for the user to select productions. When the user requests that a production be applied, Cockpit updates the graph. When the user asks that a task be *executed*, Cockpit sends a message to the task's manager program. For terminal tasks, the manager responds by invoking the tool. For non-terminal tasks, the manager responds by using encoded knowledge to select one or more productions. Cockpit then applies the productions and sends a message to each production's manager program, which executes the subtasks. The same sequence is repeated for the subtasks until terminal subtasks are reached. In this automatic mode, the process proceeds without designer intervention. The designer may, however, reverse any decision made by a manager program.

When the output of a task is not satisfactory, it is necessary to backtrack. Either different parameters must be supplied to some of the tools, different tools must be chosen, or a task must be decomposed in an entirely different way. The designer directs Cockpit to backtrack to before a certain production was applied. Cockpit adjust the display so that the designer can start over at that point and applying another production to the same non-terminal task node. Cockpit creates a new design state before backtracking in case the designer later changes his mind.

In automatic mode, a hierarchy of managers is created. Some of the managers are responsible for executing tasks, while others execute particular productions. In the manager hierarchy, all of the children of a task manager are production managers and all of the children of a production manager are task managers. Each production

manager has one child manager for each subtask (task node on the right hand side of the production). Task managers have multiple children when several productions are attempted, or the same production is attempted repeatedly.

In order to compute intelligent ratings and set parameters appropriately, the managers must often gather more information. A query handling protocol allows managers to request the desired information from other managers. Managers can send queries to the managers of subtasks or parent tasks. Cockpit routes these query messages to the appropriate manager, but does not interpret them. The architecture does not specify which quantities may be queried; the implementor of each manager decides what queries to send and respond to. This query mechanism may be used to find the results of previous attempts to perform this task. Decisions can be adjusted accordingly to improve upon the results, which is how iterative design processes are implemented.

Each manager program may perform the following functions:

Pre-Evaluation (production managers only) Assign a rating indicating a production's likelihood of success in the current situation.

Task Execution (task managers only) For terminal tasks, invoke whatever tools are needed. If the tool requires parameter settings, determine the parameters. For non-terminal tasks, choose productions. This function has the authority to choose multiple productions if it decides that exploring the alternatives in parallel is warranted.

Production Execution (production managers only) Issue messages to Cockpit to execute subtasks at the appropriate time. Usually, the scheduling of subtasks is simply determined by data dependencies.

Post-Evaluation (task managers only) Once the tools or subtasks have completed, check the appropriate constraints to determine whether the task was accomplished successfully.

Query Handling (both task and production managers) Respond to task specific queries by estimating quantities or forwarding the query to a child or parent manager. Implementors must negotiate with each other about what queries should be handled by each manager.

The techniques used to perform these functions are intentionally not specified as part of the architecture so that the most appropriate techniques may be used. For example, some managers may be algorithmic while others use neural networks or rule bases. The architecture simply defines a set of messages that managers are expected to be able to respond to or are allowed to send to Cockpit. Some implementation options are discussed in Section 9.3. A single manager program may encapsulate the knowledge for several tasks. Each message from Cockpit indicates what task is being evaluated or executed and provides the filenames for all of the inputs and outputs. The constraints may be included in one of the input files or may be passed to the manager by any other method the implementing programmer chooses.

9.2 Execution example

In this section, a synthesis scenario illustrates how the architecture is used. Suppose that a customer of the aircraft engine company asks for help selecting an engine and integrating it into the aircraft he is building. To start, the designer runs Cockpit with an input file indicating the standard tools and productions that are available on his site. The start graph is the one shown in Figure 8.3.

9.2.1 Manual operation

After selecting an engine, the designer decides to proceed to the task **Engine Mount Design**. Upon selecting this task, Cockpit tells him that it can be decomposed in two different ways, as indicated in Figure 9.2. Production **Mount1** calls for the engine mount design to be selected from a catalog, whereas production **Mount2** describes a procedure to design a custom engine mount. To help the designer decide which production to choose, Cockpit sends a message to the manager program of each of these productions. The manager program for **Mount1** looks at the input files and notices that there are no mounts for this airframe in the database. Therefore, there is a low probability of finding an acceptable engine mount, and the manager assigns a low rating. The rating for **Mount2** is moderate because it is very likely to succeed but requires considerable effort and often results in a more expensive mount. These ratings are displayed by Cockpit, but are only advisory. Designers can choose whichever production they prefer. In this case, the designer agrees with the ratings and ask Cockpit to apply production **Mount2**. Cockpit adjusts the display to show icons for the new subtasks. This sequence of actions is illustrated in Figure 9.3.

9.2.2 Query handling

Similarly, when the designer selects the icon for **Mount Ring Selection**, Cockpit sends a pre-evaluation message to both of the productions in Figure 9.4. Often, a manager can make use of information that is not in the input specifications in assigning a more accurate rating. In this case, the manager of production **RingSel1** knows that the automatic ring selector tool usually does a good job, but does not make use of stress information. **RingSel1** should normally get a high rating because it runs quickly with little user intervention. However, if stress in the ring is known to be a problem, the production would get a low rating. The manager determines

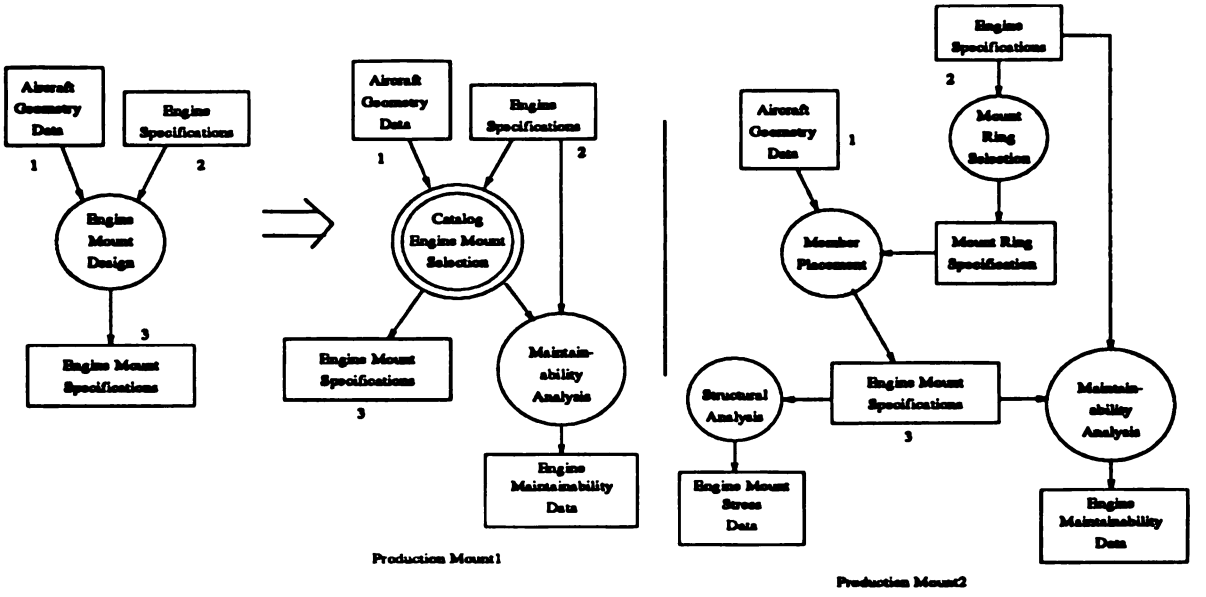


Figure 9.2. Productions for **Engine Mount Design**

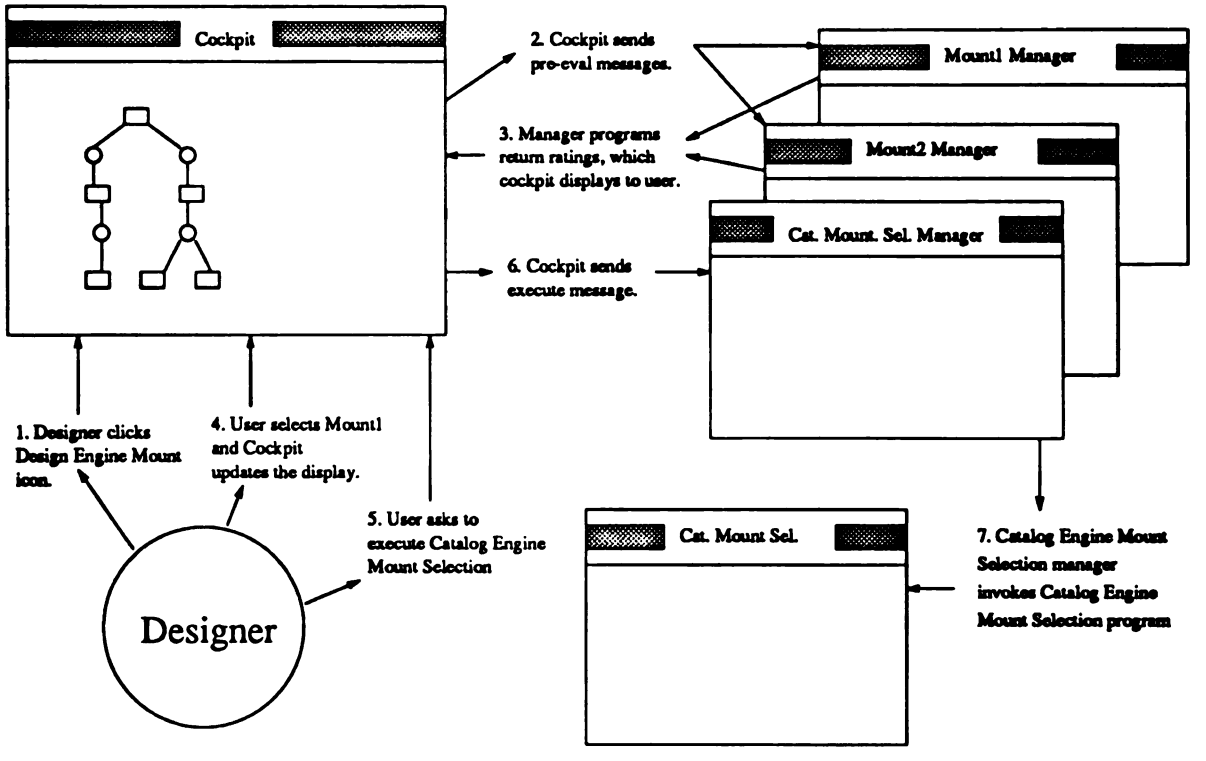


Figure 9.3. Sequence of actions during manual operation

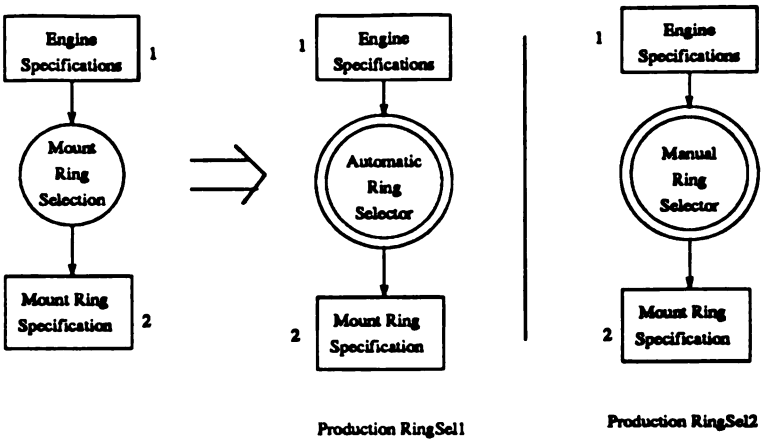


Figure 9.4. Productions for `Mount Ring Selection`

whether stress is a problem by sending a query to its parent, which is the manager of the `Mount Ring Selection` task. That manager forwards the query to its parent, the manager of the production `Mount2`. That manager recognizes that it can handle the query by looking at the stress data produced from previous attempts. Since there have been no previous attempts, no stress data is available. The manager replies to the query, indicating that stress is not known. Cockpit routes this reply back to the manager of `RingSel1` which uses the information in computing its rating. This sequence of actions is illustrated in Figure 9.5.

9.2.3 Backtracking

After a mount ring is selected, the designer performs `Member Placement`, `Structural Analysis`, and `Maintainability Analysis`. During `Maintainability Analysis`, a problem is discovered. While there is clearance between the oil filter and the engine mount, the clearance is not sufficient to unscrew the oil filter when it needs to be changed. The designer backtracks to `Member Placement` and places the offending part elsewhere. Then he repeats `Structural Analysis` and `Maintainability Analysis` on the new version of the design.

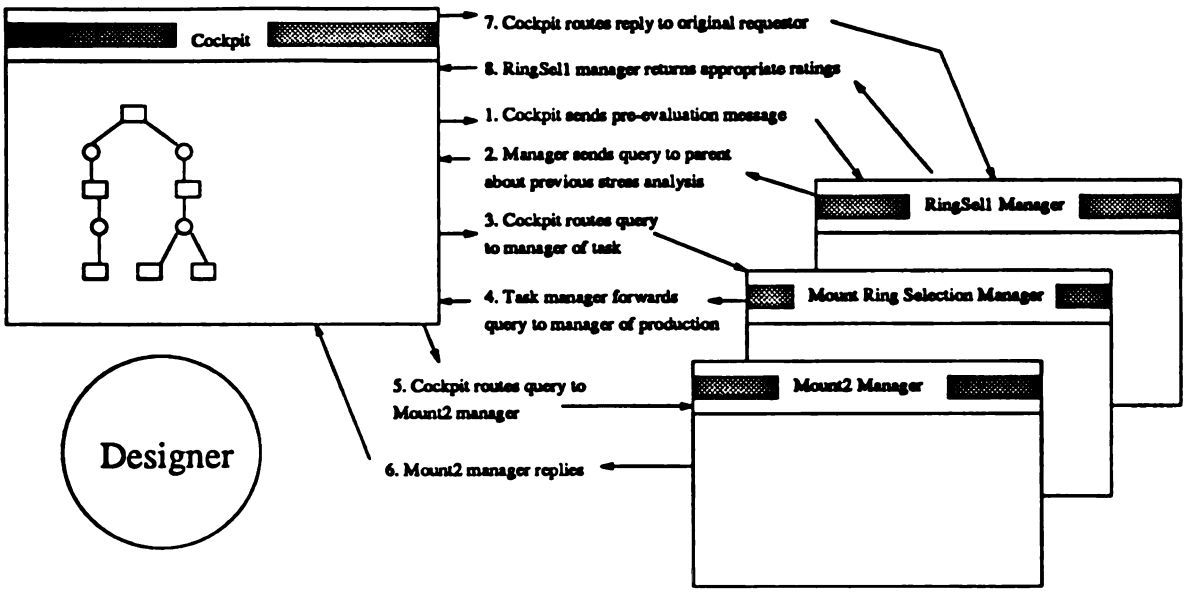


Figure 9.5. Sequence of actions for query handling

Determining where to backtrack to is not always a simple problem. In this case, the designer used domain knowledge to determine which task probably caused the problem. There may be several alternative ways of correcting the problem. For example, he could have backtracked all the way to **Engine Selection** and selected an engine with the oil filter mounted elsewhere.

This example shows an important advantage of using a methodology management system. The output of Engine Mount Design was produced after the **Member Placement** task was completed the first time. Without a methodology management system, it is tempting to consider the task done and release the design at that time. Tasks like Maintainability Analysis are easily forgotten. The problem would have shown up when the prototype was assembled, but would have been very expensive to fix at that time.

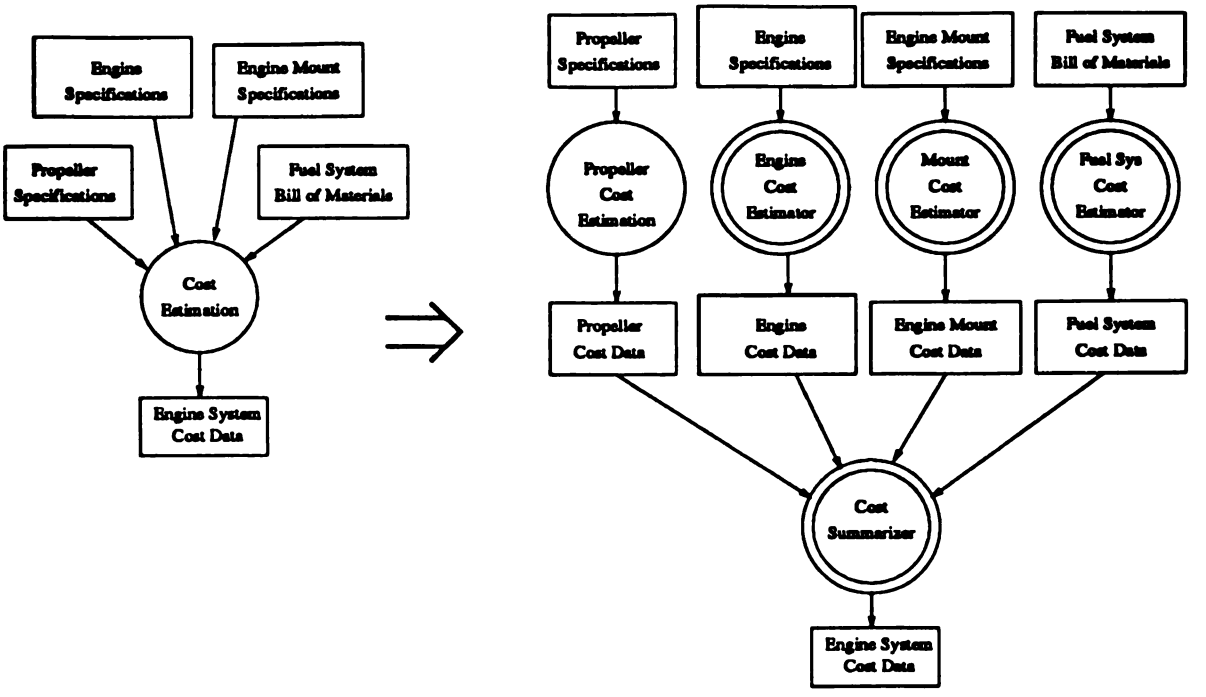


Figure 9.6. Production for `Cost Estimation`

9.2.4 Automatic operation

Later in the scenario, the designer comes to the task `Cost Estimation`. Since he trusts the knowledge encoded for this task, he uses the automatic mode. He selects the `Cost Estimation` icon and Cockpit displays the only choice. It tells him that the task can be decomposed as shown in Figure 9.6. He executes the task in automatic mode by clicking the "execute" button instead of the "apply" button. Cockpit sends an execute message to the manager for `Cost Estimation`, indicating what productions are available. Since only one is available in this case, the selection is trivial. The execute function of the `Cost Estimation` manager sends messages back requesting that the production be applied and executed.

The manager of the production then requests that the `Propeller Cost Estimation` task be executed. Cockpit then determines that the two productions shown

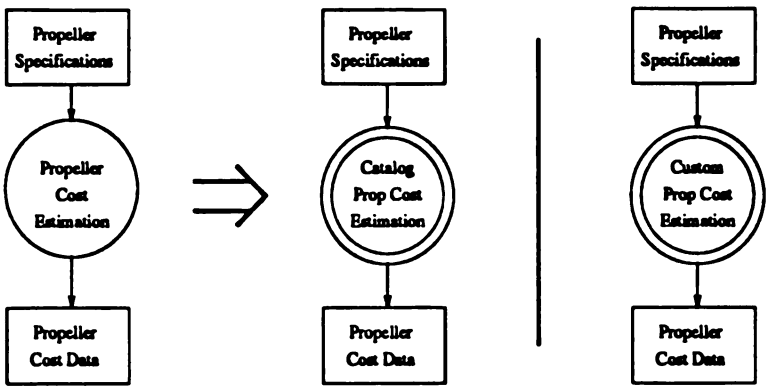


Figure 9.7. Productions for **Propeller Cost Estimation**

in Figure 9.7 could be applied. One of the productions works best for custom designed propellers, while the other only works for standard propellers which have been selected from a catalog. Cockpit gets ratings from both productions' managers and sends an execute message to the **Propeller Cost Estimation** manager (Execute messages contain the ratings.). That manager selects the higher rated production and sends messages to Cockpit asking that it be applied and executed. This sequence is illustrated in Figure 9.8. The other subtasks of **Cost Estimation** are executed in a similar fashion.

Managers of productions can execute subtasks simultaneously when there is no dependence. If a subtask fails, the production manager may chose to backtrack by re-executing an earlier subtask or may report failure to its parent. Also, if a manager of a task deems it appropriate, multiple productions can be applied and executed simultaneously. If the selected productions fail, the manager can select others or report failure to its parent. Any decisions made by a manager program while in automatic mode can be reversed by the designer if necessary.

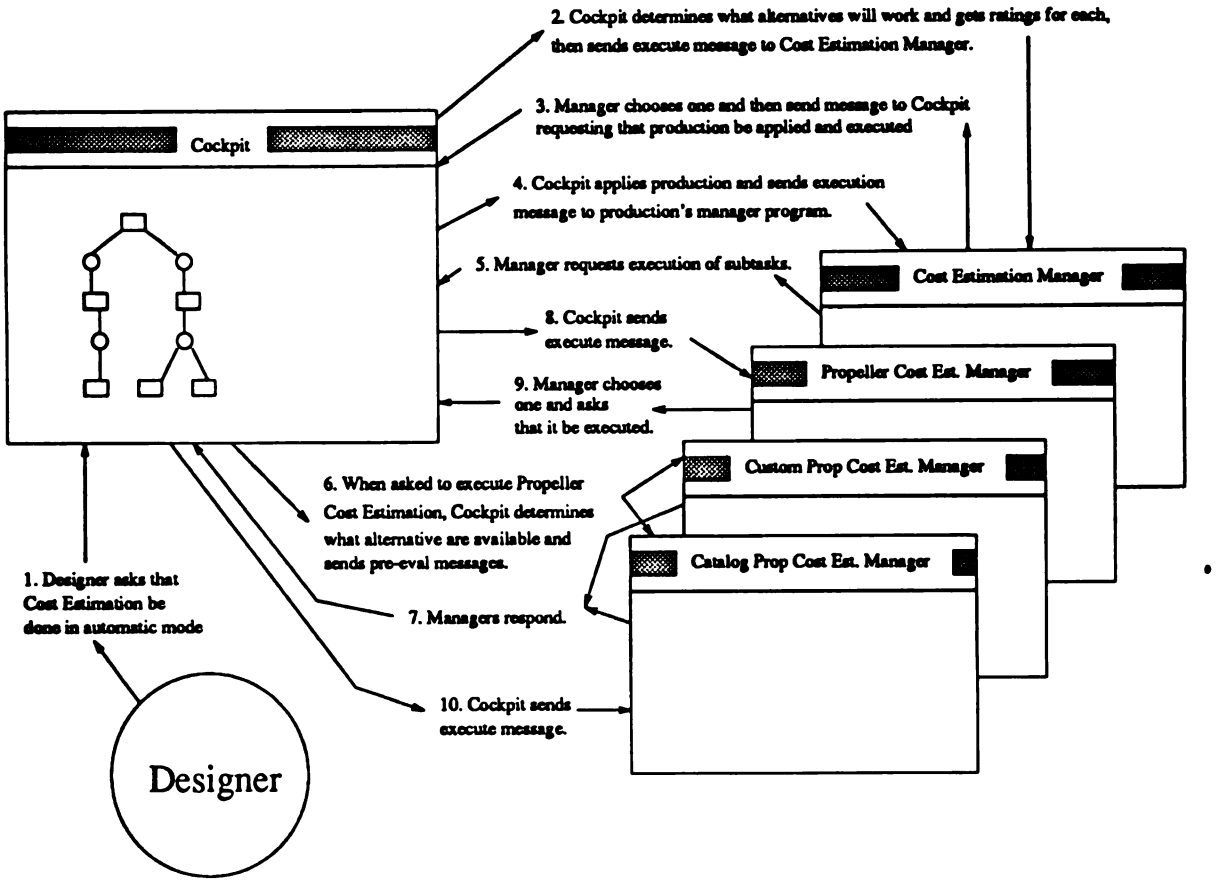


Figure 9.8. Sequence of actions during automatic operation

9.3 Implementation options for manager programs

Although the methodology management architecture does not specify how manager programs should be implemented, this section discussed some possible implementations. Since manager programs operate by responding to messages from Cockpit, an event driven programming style is most effective. Each of the responsibilities of manager programs are addressed separately.

Ideally, a system integrator would be able to specify the behavior of each manager in a data file using a convenient notation. Then, either source code could be produced automatically from the file or a generic manager program could interpret the data file at run time. However, whatever notation is chosen, there will probably be some desired behavior that cannot be expressed. Therefore, it may occasionally be necessary to modify the source code of manager programs manually. Even when manager programs must be customized, other manager programs can easily be used as templates, implementing the majority of the functionality.

9.3.1 Tool invocation

The manager of a terminal task node should respond to a TASK_EXECUTE message by invoking the appropriate application program and reporting success or failure when it completes. The filenames (or other identifiers) of inputs and outputs are included in the TASK_EXECUTE message. In many cases, it would be sufficient to use a predetermined command line that is stored in a data file. In other cases, however, the manager must determine parameters of the tool using task specific knowledge. In these cases, custom manager programs would be necessary.

9.3.2 Abstract task execution

The manager of a non-terminal task node should respond to a TASK_EXECUTE message by choosing one or more productions. The TASK_EXECUTE message from cockpit contains a list of candidate productions, sorted by the ratings that their managers assigned to them. The manager can apply as many of these as it chooses. When the production has been executed, Cockpit either sends a PROD_FAILED or PROD_SUCCESS message to the manager. The manager can explicitly request re-evaluation later. The results, reported in EVALUATION_REPORT messages, may be different due to information generated by productions that have been executed.

One possible manager implementation would use the automata shown in Figure 9.9. The various states indicate how many productions are currently executing and whether a re-evaluation is pending. This manager executes at most two productions at a time, but could easily be extended to run an arbitrary number. It continues trying until there are no productions with a pre-evaluation above a threshold value. If enough resources are available, the manager attempts two productions simultaneously. If any production succeeds, it immediately checks any constraints and, if passed, reports success to Cockpit. This approach does not require any task specific knowledge, so it could easily be included in a generic manager program.

9.3.3 Production execution

Another responsibility a manager program can have is production execution, which is simply the task of executing the subtasks in an appropriate order. Cockpit requests that the production be executed by sending a PROD_EXECUTE message. When a subtask has completed, Cockpit sends a SUBTASK_FAILURE or SUBTASK_SUCCESS message to the manager.

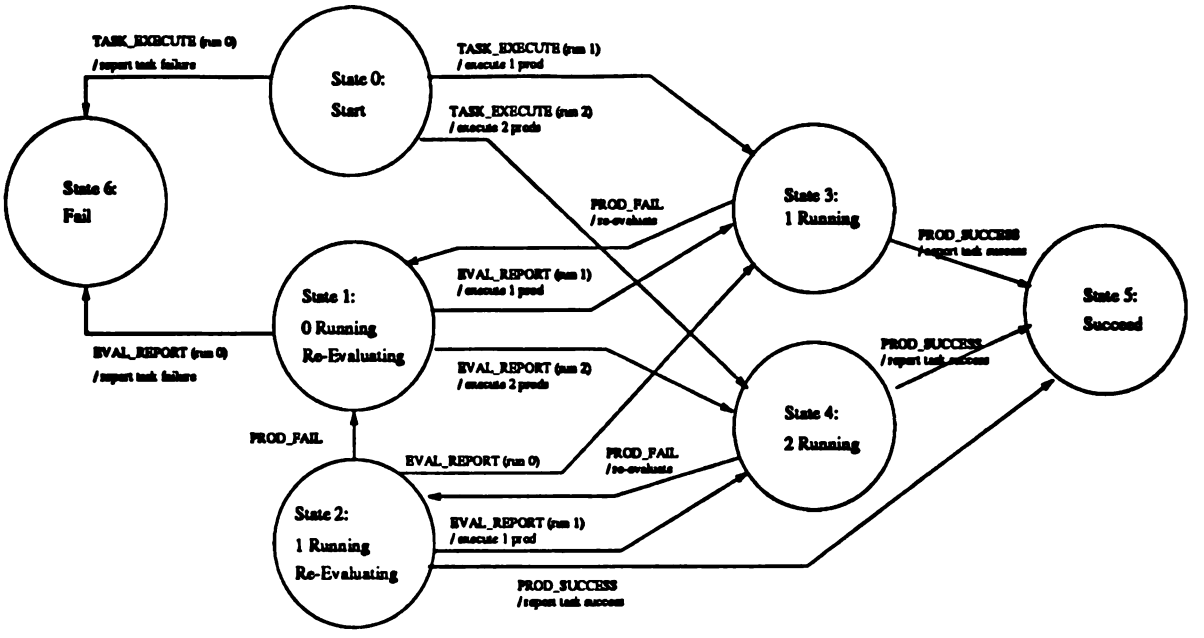


Figure 9.9. Task Execution Automata

Figure 9.10 shows an automata for managing the production **Mount2**. The various states indicate the current status of each subtask. The first two subtasks must be executed in series, while the later two may be executed in parallel (see Figure 9.2. If any subtask fails, it immediately reports failure to Cockpit. If all subtasks succeed, the manager checks any constraints and, if the constraints are satisfied, reports success to Cockpit. The only task specific knowledge in this function comes directly from the production. A similar manager could be generated automatically from any other production. Task specific knowledge could be used to improve this function, for example, by finding a way to re-execute either **Ring Selection** or **Member Placement** to correct a problem when **Structural Analysis** or **Maintainability Analysis** fails.

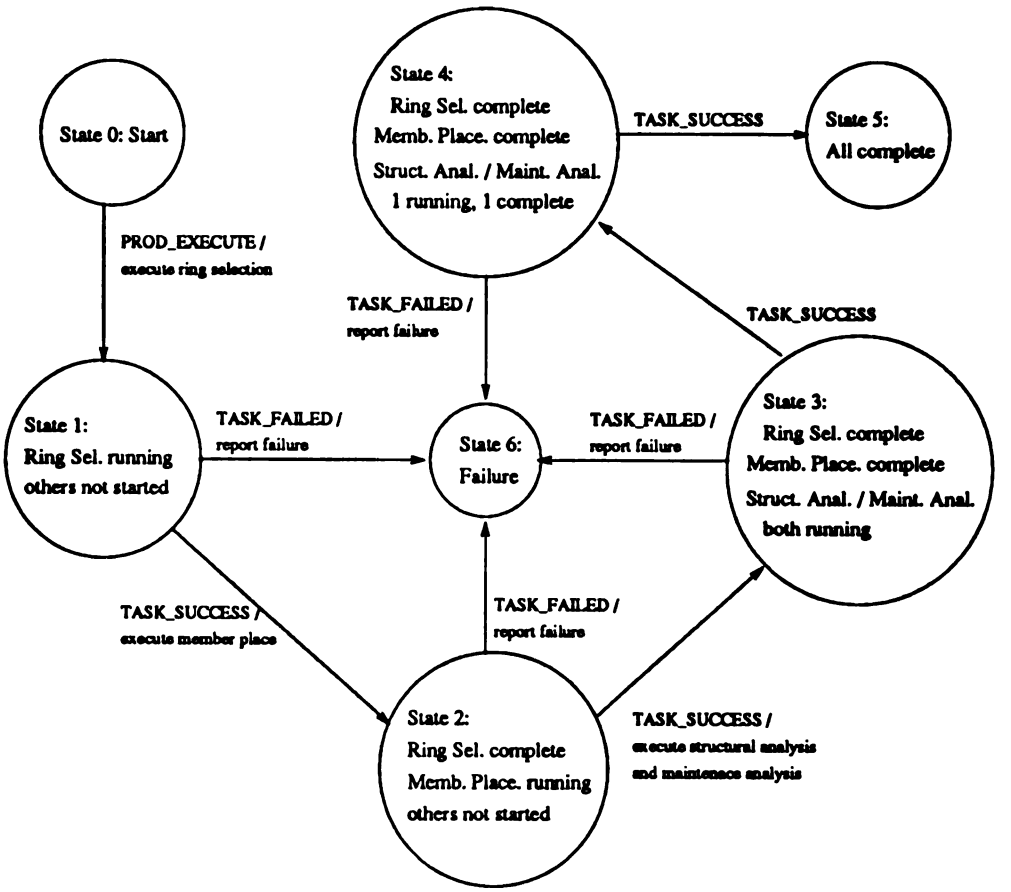


Figure 9.10. Production Execution Automata for `Mount2`

9.3.4 Pre-evaluation

The functions used to assign ratings to productions will differ widely. In some cases, it may be sufficient to assign ratings statically based on which productions have succeeded most often in the past. These static ratings could be adjusted downward if the production has been tried unsuccessfully on this task node already (which could be determined using the query mechanism). Alternatively, the ratings may be a function of parameters obtained through the query mechanism or by sending framework messages (as discussed in Chapter 2 to input specification nodes. Sophisticated pre-evaluation functions may gather metrics continuously about what conditions lead to success and adjust their ratings accordingly.

9.3.5 Post-evaluation

After a task finishes, the manager needs to check for success. It may be possible to express the criteria for success using constraints as described in Chapter5. If so, these constraints could be stored in a simple data file. If more sophisticated checking is required, custom manager programs must be written.

9.3.6 Query handling

Useful query handling almost always requires significant task specific information. In some cases, a data file similar to a `frame` representation (see chapter 4) may be a sufficient mechanism for indicating how to compute the needed information. In many situations, however, customization is needed.

Part V

Conclusion

CHAPTER 10

Conclusion

A CAD framework has been described which would make designers more productive and reduce design errors by:

- Organizing the design data such that important realationships are recorded. Relationships include representations that describe different aspects of the same object, derivation histories, component relationships, and generalization / specialization relationships.
- Effectively modeling product families that have many different variants determined by customer selected options. Designs can conveniently specify what combinations of optional features are to be offered and how the products structure and performance varies depending on the options chosen.
- Providing services which utilize information from multiple sources and therefore cannot be provided by individual application tools. These services include computing design properties, checking constraints, selecting components such that constraints are satisfied, and notifying designers of changes that influence them.
- Guiding designers through appropriate design methodologies such that important tasks are not overlooked or delayed.

10.1 Implementation status

The data management and services functionality was implemented separately from the methodology management functionality. Both were implemented as single user, single processor systems.

The following functionality was completely implemented:

- Discipline independent data model
- Assembly Representations
- Parser for language defined in appendix B
- Most algorithms for manipulating MDDs
- Frame Representations

The following functionality is partially implemented:

- Constraint checking - Implementation works but is not optimized for efficiency. Only Algorithm check_constraint_A is implemented. The implementation does not handle environment constraints differently, as it should.
- Cockpit Program - Manual mode is implemented, but automatic mode is not.
- Manager Programs - Automatic mode not implemented. Pre-evaluation functions do not use any task specific knowledge.

The following functionality was not implemented as part of this research:

- Automated component selection - Only feature implemented is ability to print combination satisfying all constraints. User must define supplemental variables and form temporary bindings manually.

- Change notification - This functionality has a low priority in a single user system.

- Release Controls - Currently, anyone is allowed to release designs into any workspace. More restrictions would be needed in a multi-user system.

10.2 Contributions

This research has extended the theoretical foundation of CAD frameworks, making more effective discipline specific and interdisciplinary CAD frameworks more likely in the future. Specifically, this research has made the following contributions:

- Defined and implemented a discipline independent, extensible data model.

- Defined a powerful language for expressing constraints and bindings and developed and implemented algorithms for evaluating them.

- Defined and implemented powerful features for specifying and analyzing designs with extensive optional content.

- Defined a flexible means for designers to express formulas for design properties and implemented a system to apply these formulas to compute property values.

- Developed an algorithm for automatic component selection with optional content. Previous solutions do not address products or components with options. The algorithm is efficient for many practical problems.

- Defined a formal representation of design methodologies using graph grammars.

- Defined and partially implemented an architecture which guides the designer through a design exercise and automatically performs design tasks.

10.3 Future work

To capitalize on this research, it is recommended that the following topics be investigated in the future:

- A distributed implementation of both data management and methodology management functionality supporting multiple users.
- Discipline specific frameworks which extend the discipline independent functionality defined in this thesis.
- Manager programs that encode realistic design knowledge.
- Methods for creating manager programs easily from convenient notations which express desired manager program behavior.
- Extension of the methodology management functionality to handle dependencies other than data dependencies between tasks, such as restrictions that two tasks be run simultaneously or run in a specific order not determined by data dependencies.
- Algorithms, similar to those implemented within frames, which solve equations simultaneously to compute property values.
- Constraint checking algorithms that can reason with the constraints on component designs to determine whether satisfying the components' constraints guarantees that the constraints of the main design are satisfied.

APPENDICES

APPENDIX A

Glossary

Abstract Task A step in a design process for which the work to be done is understood, but the actual tools are not specified. See Chapter 8.

Applicability Set The set of option combinations offered for a design. See Chapter 3.

Assembly Representation A representation type, provided as part of the discipline independent system, that represents the structure of a design. See Chapter 2.

Binary Decision Diagram See also Ordered Binary Decision Diagram.

Binding A reference to another design from within a representation. See Chapter 2.

Case Label Part of a generalized binding that indicates when a particular case applies. See Chapter 3.

Check-in / Check-out A version control paradigm in which objects are modified outside the system and checked in (entered into the system) at various times. Specific versions can then be checked out (copied to an external place). See Chapter 2.

Checkpoint A configuration that cannot be modified. Checkpoints are used to record the state of an entire component hierarchy for later reference. Other authors use the term snapshot. See Chapter 2.

Classification Hierarchy A hierarchy in which each object has an "is a type of" relationship with its ancestors. See Chapter 2.

Cockpit Program A program in the proposed methodology management system. Cockpit records the current status of the design process and interacts with the user and a set of manager programs. See Chapter 9.

Compatibility Suitability for combined use. In the context of this dissertation, representations are compatible if they are not based on contradictory assumptions or information. See Chapter 8.

Completeness In the context of this dissertation, completeness is a property of design process grammars indicating that there are sufficient products in the grammar to guarantee success of design planning. See Chapter 8.

Component Hierarchy A hierarchy in which each object has an "is part of" relationship with its ancestors. See Chapter 2.

Conditional Inclusion A construct used to model products with optional content. A conditionally included portion of a representation should be ignored when the specified Boolean expression of option variables is false. See Chapter 3.

Configuration In this dissertation, a description of a component hierarchy at a given time. In [14], one version of each of a group of related files. In [22], one version of each of a set of designs. In [1], an association between a composite object and a version of each of the composite object's components. See Chapter 2.

Constraint A condition which must be satisfied for a design to be considered acceptable. See Chapter 5.

Derivation Hierarchy A hierarchy in which each object has an "is derived from" relationship with its ancestors. See Chapter 2.

Design Object A type of object in the proposed data model which keeps track of all of the versions for a particular design. Design objects also record the option variables, option restrictions, and constraints that apply to any of its versions. See Chapter 2.

Design Process Grammar A proposed formalism for representing sets of acceptable design processes. See Chapter 8.

Design State A single, consistent (though possibly partial) description of an artifact being designed. See Chapters 8 and 2.

Design Version A type of object in the proposed data model which partially represents a design state. The states of component designs are represented elsewhere. See Chapter 2.

Designer A person that contributes to the specification or analysis of a product.

Dynamic Binding A type of binding in which the referenced version is determined when the binding is evaluated, as opposed to when the binding is defined. See Chapter 2.

Environment Constraint A constraint that restricts the properties of the environment in which a design can be used. See Chapter 5.

Equivalence Relationship The relationship among representations that describe different aspects of the same artifact. See Chapter 2.

Frame A representation type, provided as part of the discipline independent system, that contains formulas for computing properties of a design. See Chapter 4.

Generalized Binding A type of binding which allows different bindings to be applied based on what options are selected. See Chapter 3.

Graph Production A rule which allows one subgraph to be replaced by another. See also Design Process Grammar. See Chapter 9.

Interest A construct used to specify what events should trigger a notification and what form the notification should take. See Chapter 7.

Manager Program Part of the proposed methodology management system. Manager programs encapsulate task specific knowledge and make this knowledge available to the designer and other manager programs by interacting with Cockpit. See Chapter 9.

Multiway Decision Diagram (MDD) A graphical data structure which efficiently represents Boolean, string valued, and numeric valued functions of enumerated variables. See Chapter 3.

Non-terminal MDD Node A type of node in an MDD. See Chapter 3.

Non-terminal Task Node A node in a process flow graph representing an abstract task. See Chapter 8.

Option Selection Object A type of object in the proposed data model that allows users to specify a subset of the applicability set for analysis. See Chapter 3.

Option Variable A construct in the proposed data model that allows users to designate a feature or property as customer selectable. Option variables may be either Boolean, string valued (with an enumerated set of possible values), or numeric (with an enumerated set of possible values). See Chapter 3.

Optional Content Features of properties of a product for which the customer is offered choices. See Chapter 3.

Ordered Binary Decision Diagram (OBDD) A graphical data structure for representing Boolean functions of Boolean variables. See Chapter 3.

Predecessor Task A step in a design process which must be completed before another (to which is a predecessor). See Chapter 9.

Primary Region The nodes of an MDD used for component selection that correspond to regular (not supplemental) option variables. See Chapter 6.

Process Flow Graph A proposed formalism which represents a particular design methodology (although the methodology may contain abstract tasks). See Chapter 8.

Product Family The collection of variants produced by enumerating all of the different option combinations in the applicability set of a design. See Chapter 3.

Production Manager A manager program that manages a production (as opposed to a task). Production managers assign ratings, schedule the execution of subtasks, and handle queries. See Chapter 9.

Production See also Graph Production.

Representation An object in the proposed data model that directly describes some aspect of an artifact. See Chapter 2.

Representation Type A particular data format for a particular type of design data. See Chapter 2.

Release Make available for use for certain purposes. See Chapter 2.

Slots References from design version to representations indicating which representation describes a particular aspect a design state. See Chapter 2.

Snapshot See also Checkpoint.

Static Binding A type of binding in which the referenced version is determined when the binding is defined, as opposed to waiting until it is evaluated. See Chapter 2.

Subtask A task represented by one of the task nodes on the right hand side of a graph production in a design process grammar. See Chapter 8.

Successor Task A step in a design process which must be completed after another (to which is a successor). See Chapter 9.

Supplemental Option Variable An additional option variable introduced by the proposed component selection algorithm. See Chapter 6.

Supplemental Region S The nodes of an MDD used for component selection that correspond to supplemental option variables. See Chapter 6.

Task A step in a design process. See Chapter 8.

Task Manager A manager program that manages an a task (as opposed to a production). Task managers invoke tools (terminal tasks only), select productions (non-terminal tasks only) and handle queries. See Chapter 9.

Terminal MDD Node A type of node in an MDD. See Chapter 3.

Terminal Task A node in a process flow graph representing a particular tool invocation. See See Chapter 8.

Tool Invocation An execution of an application program. See Chapter 8.

Variant The design corresponding to a particular combination of option choices for a design that has optional content. See Chapters 1 and 3.

Version A description of a particular design concept at a particular time. See Chapter 2.

Versioned Flow Graph A variation of a Process Flow Graph that allows multiple alternatives for an abstract task to coexist. See Chapter 8.

Workspace In this dissertation and in [21], a mechanism for selecting version when evaluating dynamic bindings. In [1], a logical data base. In [22], a collection of versions of a set of objects, possibly including more than one version for some objects. In [17], an object which stores process related meta-data. See Chapter 2.

APPENDIX B

Language Constructs

B.1 Overview

Throughout this thesis, a special language has been used to express option restrictions, case labels in generalized bindings, formulas, and constraints. The language design objectives are to express all of the required information using syntax that is natural for a designer and that may be rapidly parsed.

The proposed language includes constructs for numeric, Boolean, and string expressions. Option restrictions and case labels must depend solely on the options chosen and can be formed using any of the following constructs:

- Numeric expressions may be constructed from numeric option variables, numeric constants, and other numeric expressions using the operators +, −, *, and /.
- String expressions may be constructed from string option variables, string constants, and other string expressions using the operator + (concatenate).
- String expressions may be constructed by enclosing a numeric or Boolean expression between colons. The result is computed by evaluating the expression and printing the result in the standard fashion.

- Boolean expressions may be constructed from Boolean option variables and other Boolean expressions using the operators + (AND), * (OR), ' (NOT), ^ (XOR), and − > (implies, A − > B is shorthand for NOT(A) OR B).

- Boolean expressions may be constructed by comparing numeric expressions using the operators =, ! = (not equal), <, >, <=, or >= or comparing string expressions using the operators = and ! =.

- Boolean expressions may be constructed by testing a numeric or string expression for membership in a set of expressions using operators = and ! =.

The syntax for a static binding is

&<design object name>.v<version number>,<string expression list>.

As discussed in Section 3.3, the string expression list is used to indicate the options for the referenced design. Similarly, the syntax for a dynamic binding is

&<design object name>,<string expression list>.

The syntax for a generalized binding is

FOR {<case list>}.

The syntax for each item in the case list is

<Boolean expression> USE <binding>;.

Constraints and formulas in frames require access to the message handling system and special access to assembly representations. Therefore, they may use any of the constructs above plus the following:

- Expressions of the form real(<string expression>) are evaluated by evaluating the string expression and sending the result as a message to the design version, expecting a numeric result. For example, if the design version has

a solid model in one of its slots, an expression for the volume would be `real(''get volume'')`. Similar string and Boolean expressions can be formed as string(<string expression>) and bool(<string expression>).

- Expressions can be simple property values. Property values can be of type string, numeric, or Boolean with the type determined by the first character of the name (like option variables). To distinguish properties from option variables, the first letter in a property name is always lower case, whereas the first letter of an option variable name is always capitalized. The value for a property is determined by sending the message "compute <name>" to the design version. For example, the subexpression `volume` would appear within an expression for the mass of an object. Frame representations allows users to enter formulas for computing property and then responds to "compute <name>" messages by evaluating the formulas.

- Expressions of the form @ <component name> . <parameter name> are evaluated by accessing the list of named parameters associated with the named `component` object. For example, the x coordinate of the battery would be `@battery.x`. A special form is \ . <parameter name> which accesses the `component` object that instantiated the current design.

- Expressions of the form @ <component name> -> <expression> are evaluated by computing the result of the expression with respect to a component design. For example, the capacity of the battery would be `@battery -> capacity`. This type of expression differs from the previous in that a property of the component's design is requested instead of a parameter of the `component` object which instantiates it. A special form is \ -> <expression> which evaluates the expression with respect to the design that instantiated the current design.

This special form is used for properties describing the environment in which something is being used such as `\-> temp` for temperature.

- Expressions of the form ~ <slot name> -> <expression> are evaluated by computing the result of the expression using the representation in the specified slot. These expressions are needed if a design version has more than one representation capable of handling a particular message. For example, the expression `~process -> cost` would send the message "compute cost" only to the representation describing the manufacturing process instead of trying all of the representations in order.

- Expressions of the form <binding> -> <expression> are evaluated by computing the result of the expression with respect to the design version referenced by the binding. It is convenient to model materials as designs and access their properties with this type of expression. For example, the elastic modulus of aluminum might be expressed as `&aluminum -> E`.

- A Boolean expression of the form @ <component name> = <binding> evaluates to true if the binding for the named component evaluates to the same design version as the binding. For example, the constraint `@Head = P10_100` restricts the choice of designs for the component Head.

B.2 Detailed syntax

These are the lexical analysis rules in lex (or flex) format:

```
int [0-9]+
dreal ([0-9]*"."[0-9]+)
ereal ([0-9]*"."[0-9]+[eE][+-]?[0-9]+)
numb {dreal}|{ereal}
smid [a-z][a-zA-Z0-9_]*
capid [A-Z][a-zA-Z0-9_]*
```

```
anyid {smid}|{capid}
str     \'[^\']*\'

%%

[ \t\n] ;
TRUE   { return TRUE; }
FALSE   { return FALSE; }
for   { return FOR; }
use   { return USE; }
string   { return STRING; }
num   { return REAL; }
bool   { return BOOL; }
{str}   { return STRING; }
{numb}   { return NUMBER; }
{int}   { return INTEGER; }
&{anyid}  { return DOBJ; }
{smid}    { return N_PARAM; }
{capid}   { return N_OPTION; }
${smid}   { return S_PARAM; }
${capid}  { return S_OPTION; }
\%{smid}  { return B_PARAM; }
\%{capid} { return B_OPTION; }
\\   { return COMP; }
@{anyid}  { return COMP; }
~{anyid}  { return SLOT; }
\-\>      { return ARROW; }
\.v       { return DOTV; }
\<   { return LT; }
\<\=   { return LTE; }
\>   { return GT; }
\>\=   { return GTE; }
\=\=   { return EQ; }
\=   { return EQ; }
\!\=   { return NEQ; }
.   { return yytext[0]; }
```

These are the parsing rules in yacc (or bison) format:

```
value_list: value_list ',' value_choice
        |
        value_choice
;

value_choice: str_expr
```

```
        |
        num_expr
;


bind: DOBJ ',' opt_str_expr_list
        |
DOBJ
        |
DOBJ DOTV INTEGER ',' opt_str_expr_list
        |
DOBJ DOTV INTEGER
        |
FOR '{' gen_case_list '}'
        ;

gen_case_list: gen_case_list gen_case
        |
gen_case
;

gen_case: opt_bool_expr USE bind ';'
        ;

str_expr: S_OPTION
        |
S_PARAM
        |
STRING
|
SLOT ARROW S_PARAM
|
STRING '(' str_expr ')'
|
SLOT ARROW STRING '(' str_expr ')'
|
bind ARROW str_expr
|
COMP ARROW str_expr
|
SLOT ARROW COMP ARROW str_expr
|
COMP '.' S_PARAM
|
SLOT ARROW COMP '.' S_PARAM
```

```
        |
bind ARROW str_expr
|
str_expr '+' str_expr
|
'(' str_expr ')'
|
':' num_expr ':'
|
':' bool_expr ':'
;

num_expr: NUMBER
        |
INTEGER
|
N_OPTION
|
N_PARAM
|
SLOT ARROW N_PARAM
|
REAL '(' str_expr ')'
|
SLOT ARROW REAL '(' str_expr ')'
|
bind ARROW num_expr
|
COMP ARROW num_expr
|
SLOT ARROW COMP ARROW num_expr
|
COMP '.' N_PARAM
|
SLOT ARROW COMP '.' N_PARAM
        |
bind ARROW num_expr
|
num_expr '+' num_expr
|
num_expr '-' num_expr
|
num_expr '*' num_expr
|
num_expr '/' num_expr
```

```
|
'-' num_expr %prec UMINUS
|
'(' num_expr ')'
;

bool_expr: B_OPTION
|
TRUE
|
FALSE
|
B_PARAM
|
SLOT ARROW B_PARAM
|
BOOL '(' str_expr ')'
|
SLOT ARROW BOOL '(' str_expr ')'
|
bind ARROW bool_expr
|
COMP ARROW bool_expr
|
SLOT ARROW COMP ARROW bool_expr
|
COMP '.' B_PARAM
|
SLOT ARROW COMP '.' B_PARAM
        |
bind ARROW bool_expr
|
bool_expr '+' bool_expr
|
bool_expr '*' bool_expr
|
bool_expr '^' bool_expr
|
'(' bool_expr EQ bool_expr ')'
|
bool_expr '''
|
bool_expr ARROW bool_expr
|
'(' bool_expr ')'
```

```
|
'(' num_expr LT num_expr ')'
|
'(' num_expr LTE num_expr ')'
|
'(' num_expr GT num_expr ')'
|
'(' num_expr GTE num_expr ')'
|
'(' num_expr EQ num_expr ')'
|
'(' num_expr EQ '[' num_expr ',' num_expr ']' ')'
|
'(' num_expr EQ '{' num_expr_list '}' ')'
|
'(' num_expr NEQ '{' num_expr_list '}' ')'
|
'(' num_expr NEQ num_expr ')'
|
'(' str_expr EQ str_expr ')'
|
'(' str_expr EQ '{' str_expr_list '}' ')'
|
'(' str_expr NEQ '{' str_expr_list '}' ')'
|
'(' str_expr NEQ str_expr ')'
|
'(' COMP NEQ bind ')'
|
'(' COMP EQ bind ')'
;

num_expr_list: num_expr_list ',' num_expr
        |
num_expr
;

str_expr_list: str_expr_list ',' str_expr
        |
str_expr
;
```

APPENDIX C

Persistent Storage Implementation

C.1 Motivation

When the implementation of this work was initiated, it was believed that a commercial object oriented data base management system would be available. However, it was necessary to implement an interim persistent storage mechanism so work could begin on implementation of the discipline independent layer. This interim system is still in use.

The requirements of the interim system were much simpler than a true object oriented database. Specifically,

- The data sets used during development would contain no more than a few hundred kilobytes of data. This data could be re-generated by a program following a schema change, so no schema update functionality was needed in the interim system. Administrative functions, such as backup and recovery, were not needed.

- The data would be accessed from one machine by a single user, though possibly by several processes. The user takes responsibility for not modifying data from two programs simultaneously, so no object locking is necessary.
- Dynamic function binding (the generic name for C++ virtual functions) must be supported for persistent objects.
- The interface must be such that migration to an object oriented database is straight forward.

C.2 Overview

Unix shared memory segments are used as the storage medium, which simplified the implementation. Unix shared memory segments survive after the processes that create them exit, providing the necessary persistence. These segments may be attached to other processes' address spaces using provided library routines. Once the segment is attached, persistent objects may be accessed by pointers in the same fashion as non-persistent objects. To ensure that a pointer always points to the same object, the shared memory segment must be attached at the same address in all processes.

A special process, called the persistence server, allocates and deallocates space in the shared memory segment. User programs request and release space by sending messages via a Unix message queue. A first fit allocation strategy is used. The data about what space has been allocated is intermixed with the actual data, proved to be a problem because errors in user programs could corrupt the space usage data and cause symptoms that were very difficult to trace. In addition to allocating and deallocating space, the server can write the contents of the segment to a disk file and read it back again. By writing the data to a file, users can remove the segment and recover the data several days later.

One of the unresolved issues in object oriented data base research is how user programs should locate persistent objects that exist before startup. Most commercial systems include some form of query processing. The approach taken in the interim system, which is also used by several persistent programming languages, is to designate one object as the root object. That object should contain enough information for the program to navigate to whatever objects are of interest. The server program keeps track of the address of the root object and provides it to user programs on demand.

Another unresolved issue is how user programs indicate whether the objects being created should be persistent or transient. Some systems add a keyword to the language. A less flexible mechanism that does not require changes to the language is to have a special persistent class. All instances of this class and any of its sub-classes are persistent. This later approach is suitable for this application and was chosen because it was much easier to implement.

C.3 The server process

The server is started from the command line, usually in the background. There should be only one server process running at a time. Upon startup, the server creates the shared memory segment and creates a message queue for incoming requests. An optional command line argument indicates the name of a file in which previously created data resides. If this argument is provided, the server reads the actual data into the segment and retrieves the address of the root object. If the argument is omitted, the server initializes the segment to contain no objects.

Then, the server simply retrieves messages from the message queue and handles them. The following message types are supported:

Setup This message indicates that a user program (client) wants to begin using persistent objects. The server responds with a message indicating at what address the memory should be attached and the address of the root object.

Allocate The server allocates a block of sufficient size and sends a return message indicating the address of this block. The size of the new object is passed in the message. Blocks are always allocated in multiples of four bytes to avoid alignment errors.

Deallocate The server releases previously allocated space. The address of the block to be released is passed in the message. The server can determine the size of the block from its internal tables. The return message simply acknowledges completion.

Show-Status The server calculates statistics about memory usage and prints them.

Commit The server saves the contents of shared memory to a disk file. The attach address and address of the root object are also recorded in the file. The name of the file is passed in the message. The root address can also be updated by this call.

Done The server destroys the shared memory segment and message queue and then exits. (This function can also be invoked by a kill signal.)

C.4 Client library routines

A library of routines is provided which enable user programs to interact with a server process. The following functions are included:

initialize Send a message to the server to determine attach address and root object address. Attach shared memory at indicated address. Return address of root object.

allocate Send a message to the server to allocate a block of specified size.

deallocate Send a message to the server to deallocate a previously allocated block.

commit Send a message to the server to write out the data to a specified disk file and possibly to change the address of the root object.

mem-status Send a message to the server requesting that space usage data be printed.

cleanup Send a message to the server to remove the shared memory segment and message queue and stop.

C.5 The persistent class

A `Persistent` class is used as a base class for all classes which should be persistent. This class has two member functions which override the default **new** and **delete** operators by calling the `allocate` and `deallocate` functions from the client library. Any time any of these classes are instantiated using the **new** operator, the new objects are persistent. Objects created as automatic variables or as data members of other classes are still transient.

C.6 Virtual functions

Dynamic function binding is implemented in C++ as virtual functions. Suppose that there is a class `foo` with two subclasses `foo1` and `foo2` as shown in the code below.

All three classes have member functions called `print()`. Now suppose that there is an instance of type `foo1`, called `x`, and a pointer of type `(foo *)`, called `y`, which points to `x`. If `print()` is a standard member function, `y->print()` calls the `print()` routine for `foo`, since the compiler has no way of knowing what class is actually pointed to by `y`. Programmers may prefer to call the `print()` function corresponding to the actual object's class instead of the `print()` function corresponding to the pointer's class, which must be determined at run time.

```
class foo {
 public:
  foo();
  virtual print();
};

class foo1 : public foo {
 public:
  foo1();
  virtual print();
};

class foo2 : public foo {
 public:
  foo2();
  virtual print();
};
```

Virtual functions are implemented in C++ as illustrated in Figure C.1. For each class, the compiler creates a *vtbl* which stores pointers to all of the virtual functions for that class. When objects are created, an extra (hidden) field in the object points to the corresponding vtbl (the actual class is known at that time). When a virtual function is called, the vtbl is consulted to determine the address of the correct function to invoke.

The mechanism does not work well for objects that are shared between programs. Suppose that process A in Figure C.2 creates object `x`, which is pointed to from process B by pointer `y` (The two processes do not need to be running at the same

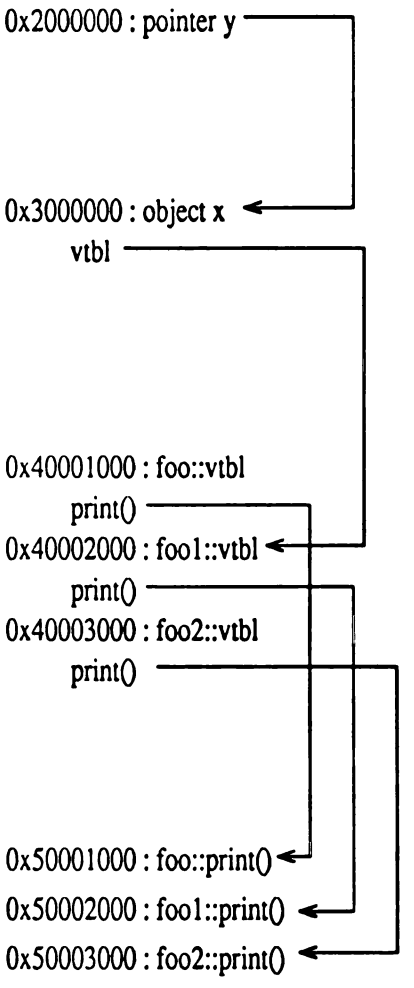


Figure C.1. Virtual functions for **transient objects**

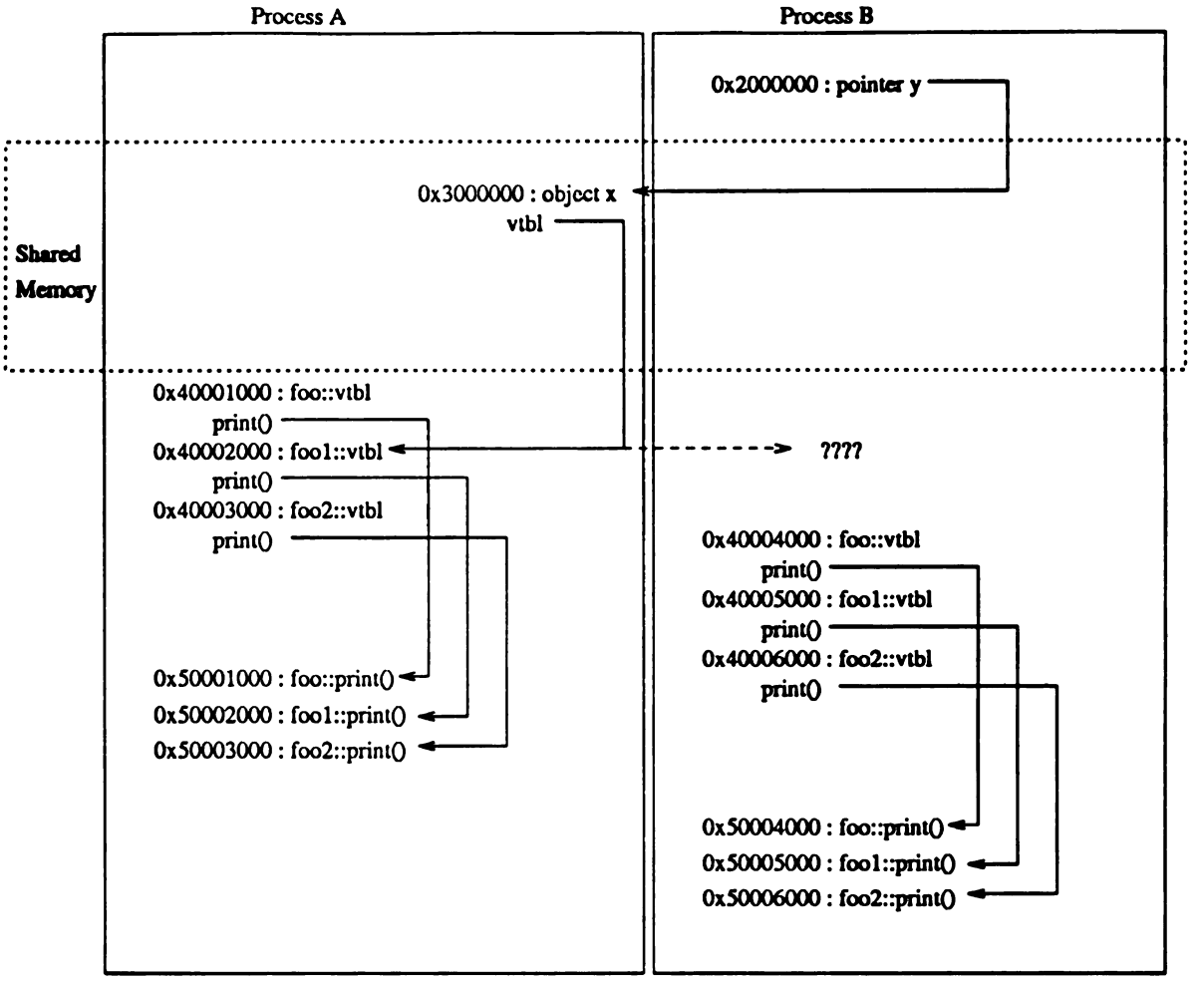


Figure C.2. Virtual functions on shared objects

time). The vtbls and `print()` functions are likely to be at different addresses in the two processes. (In this case it is usually the same program but has been modified and re-compiled.) The vtbl pointer for object **x** points to a meaningless address in process B. When `y->print()` is called, whatever happens to be at that address is treated as a vtbl and the behavior is highly unlikely to be correct.

The implementation solves this problem by adding an extra level of indirection, which must managed directly by programmers. The modified code is shown below. Each process keeps a list of pointers to *dummy objects* of each subtype. These dummy objects have correct vtbls for the program they are contained in. An integer type

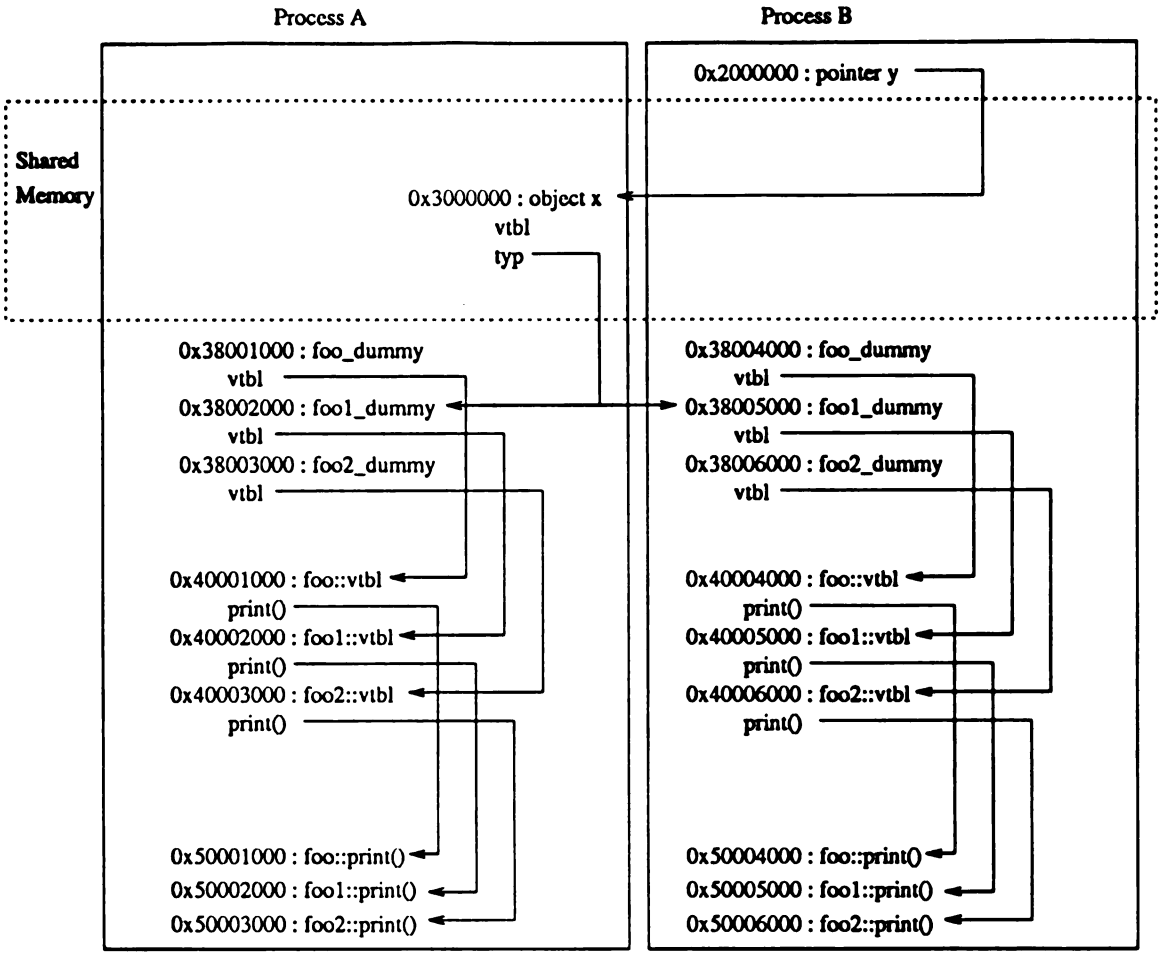


Figure C.3. Virtual Function implementation for persistent objects

identifier is added to each object as a data member. This identifier is used to find the dummy object of the same type in the list. The old `print()` functions are replaced with `print()` functions that take a pointer to the object as an explicit argument (A pointer to the object is usually a hidden argument for member functions). A non-virtual `print()` function is added to the base class. This new function simply calls the modified virtual `print()` function via the vtbl of the appropriate dummy object. The modified `print()` function assigns the explicit argument to the variable **this** (which the compiler hates) before doing the normal printing routine. This scheme is illustrated in Figure C.3.

```
class foo {
  int typ;
 public:
  foo();
  void print(); /* NOT virtual */
  virtual void print(foo *it);
};

class foo1 : public foo {
 public:
  foo1();
  virtual void print(foo *it);
};

class foo2 : public foo {
 public:
  foo2();
  virtual void print(foo *it);
};

class foo_tbl {
 public:
  (foo *) list[3];
  foo_tbl() {
    static foo foo_dummy;
    static foo1 foo1_dummy;
    static foo2 foo2_dummy;
    list[0] = foo_dummy;
    list[1] = foo1_dummy;
    list[2] = foo2_dummy;
  };
} foo_list;

void foo::print() {
  foo_list.list[typ]->print(this);
};

foo1::foo1() {
  typ = 1;
};

void foo1::print(foo *it) {
  this = (foo1 *) it;
  /* whatever would have been in foo1::print() */
};
```

This solution does produce correct behavior but is not convenient from a software maintenance point of view. Each time a new subclass is created, a new type identifier must be defined and new dummy objects must be created. Programmers adding new subclasses must be familiar with this mechanism, which has not been a major issue during the development of the prototype because only one programmer was involved. It would be completely unacceptable, however, for a production framework.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] R. H. Katz, R. Bhateja, E. E.-L. Chang, D. Gedye, and V. Trijanto, "Design version management," in *IEEE Design and Test*, pp. 12–22, 1987.

[2] P. van der Wolf, G. Sloof, P. Bingley, and P. Dewilde, "Meta data management in the NELSIS CAD framework," in *27th ACM/IEEE Design Automation Conference*, pp. 142–145, 1990.

[3] A. H. Bond and R. J. Ricci, "Cooperation in aircraft design," in *Research in Engineering Design*, vol. 4, pp. 115–130, 1992.

[4] R. A. Schmidtberg and M. A. Yerry, "Designing complex assemblies using the top-down approach," in *AUTOFACT 8*, 1986.

[5] S. B. Navathe, S. K. Murthy, and A. Cornelio, "A database approach to engineering design by selection," *Journal of Intelligent Manufacturing*, vol. 3, pp. 149–162, 1992.

[6] D. C. Brown and B. Chandresekaran, "Expert systems for a class of mechanical design activity," in *Knowledge Engineering in Computer-Aided Design*, Amsterdam: North-Holland, 1985.

[7] D. E. Whitney, "Nippondenso co. ltd: A case study of strategic product design," in *Research in Engineering Design*, vol. 5, pp. 1–20, 1993.

[8] Personal Conversation with Tim Kern, former C.E.O. of Mosler Motors, Inc.

[9] L.-C. Liu, P.-C. Wu, and C.-H. Wu, "Design data management in a CAD framework environment," in *27th ACM/IEEE Design Automation Conference*, pp. 156–161, 1990.

[10] K. ten Bosch, P. Bingley, and P. van der Wolf, "Design flow management in the NELSIS CAD framework," in *28th ACM/IEEE Design Automation Conference*, pp. 711–716, 1991.

[11] A. Wong and D. Sriram, "SHARED: an information model for cooperative product development," in *Research in Engineering Design*, vol. 5, pp. 21–39, 1993.

[12] A. P. Buchmann and C. P. de Celis, "An architecture and data model for CAD databases," in *Conference on Very Large Databases*, 1985.

[13] L. Rice, "Managing CAD/CAM/CAE data and design processes in mechanical engineering," in *International Computers in Engineering Conference and Exhibit*, pp. 421–428, 1987.

[14] W. F. Tichy, "Design, implementation, and evaluation of a revision control system," in *International Conference on Software Engineering*, 1982.

[15] M. J. Rochkind, "The source code control system," *IEEE Transaction on Software Engineering*, pp. 364–370, 1975.

[16] R. H. Katz and E. Chang, "Managing change in a computer-aided design environment," in *Conference on Very Large Databases*, 1987.

[17] S. Kim, *Configuration Managment and Version Data Modeling in VLSI Design Environments.* PhD thesis, Michigan State University, 1994.

[18] D. J. Ecklund, E. F. Ecklund, R. O. Eifrig, and F. M. Tonge, "DVSS: a distributed version storage server for CAD applications," in *Conference on Very Large Databases*, 1987.

[19] D. Beech and B. Mahbod, "Generalized version control in an object-oriented database," in *IEEE Data Engineering Bulletin*, pp. 14–22, 1988.

[20] K. R. Dittrich and R. A. Lorie, "Version support for engineering database systems," *IEEE Transactions on Software Engineering*, pp. 429–436, 1988.

[21] S. Banks, C. Bunting, R. Edwards, L. Fleming, and P. Hackett, "A configuration management system in a data management framework," in *28th ACM/IEEE Design Automation Conference*, pp. 699–703, 1991.

[22] M. Silva, D. Gedye, R. Katz, and R. Newton, "Protection and versioning for OCT," in *26th ACM/IEEE Design Automation Conference*, pp. 264–269, 1989.

[23] A. Biliris, "Database support for evolving design objects," in *26th ACM/IEEE Design Automation Conference*, pp. 258–263, 1989.

[24] E. C. VanHorn and R. R. Rezac, "Experience with the D-BUS architecture for a design automation framework," in *26th ACM/IEEE Design Automation Conference*, pp. 209–214, 1989.

[25] P. Klahold, G. Schlageter, and W. Wilkes, "A general model for version management in databases," in *Conference on Very Large Databases*, 1986.

[26] F. R. Wagner and A. H. V. de Lima, "Design version management in the GARDEN framework," in *28th ACM/IEEE Design Automation Conference*, pp. 704–710, 1991.

[27] M. Blaha, W. Premerlani, A. Bender, R. Salemme, M. Kornfein, and C. Harkins, "Bill-of-material configuration generation," in *International Data Engineering Conference*, pp. 237–244, 1990.

[28] D. P. Smolik, *Material Requirements of Manufacturing.* Van Nostrand Reinhold Company, 1983.

[29] H. Hegge and J. Wortmann, "Generic bill-of-material: A new product model," *International Journal of Production Economics*, vol. 23, pp. 117–128, 1991.

[30] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, pp. 677–691, August 1986.

[31] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macil, A. Pardo, and F. Somenzi, "Algebraic decison diagrams and their applications," in *International Conference on Computer Aided Design*, pp. 188–191, 1993.

[32] Y.-T. Lai, M. Pedram, and S. B. Vrudhula, "Fgilp: An integer linear program solver based on function graphs," in *International Conference on Computer Aided Design*, pp. 685–689, 1993.

[33] Y.-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verificiation," in *29th ACM/IEEE Design Automation Conference*, pp. 608–613, 1992.

[34] M. Minsky, "A framework for representing knowledge," in *The Psychology of Computer Vision*, New York: Ed. Winston, P.H., McGraw-Hill Book Company, 1975.

[35] D. C. Brown and R. Breu, "Types of constraints in routine design problem-solving," in *Applications of Artificial Intelligence in Engineering Problems*, 1986.

[36] A. Kott and J. May, "Decomposition vs. transformation: Case studies of two models of the design process," in *Computers in Engineering*, 1989.

[37] D. R. Brown and K.-Y. Hwang, "Solving fixed configuration problems with genetic search," in *Research in Engineering Design*, vol. 5, pp. 80–87, 1993.

[38] A. Kott, G. Agin, and D. Fawcett, "Configuration tree solver: A technology for automated design and configuration," in *Advances in Design Automation*, 1990.

[39] J. Malmqvist, "A design system for parametric design of complex products," in *Advances in Design Automation*, 1990.

[40] D. C. Brown and B. Chandresekaran, "Knowledge and control for a mechanical design expert system," *IEEE Computer*, pp. 92–100, July 1986.

[41] B. Chandresekaran, "Design problem solving: A task analysis," *AI Magazine*, vol. 11, no. 4, pp. 59–71, 1990.

[42] H.-T. Chou and W. Kim, "Versions and change notification in an object oriented database system," in *ACM/IEEE Design Automation Conference*, 1988.

[43] M. Palaniappan, N. Yankelovich, G. Fitzmaurice, A. Loomis, B. Haan, J. Coombs, and N. Meyrowitz, "The envoy framework: An open architecture for agents," *acm Transactions on Information Systems*, pp. 233–264, July 1992.

[44] D. V. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Transactions on Engineering Management*, pp. 71–74, August 1981.

[45] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, "Organizing the tasks in complex design projects," in *Design Theory and Methodology - DTM90*, 1990.

[46] D. A. Gebala and S. D. Eppinger, "Methods for analyzing design procedures," in *Design Theory and Methodology - DTM91*, 1991.

[47] P. R. Sutton, J. B. Brockman, and S. W. Director, "Design management using dynamically defined flows," in *30th ACM/IEEE Design Automation Conference*, pp. 648–653, 1993.

[48] M. Nagl, "A tutorial and bibliographic survey on graph grammars," in *Graph Grammars and their Application to Computer Science and Biology*, Berlin: Springer-Verlag, 1979.

[49] H. Ehrig, "Introduction to the algebraic theory of graph grammars," in *Graph Grammars and their Application to Computer Science and Biology*, Berlin: Springer-Verlag, 1979.

[50] H. Ehrig, "Tutorial introduction to the algebraic theory of graph grammars," in *Graph Grammars and their Application to Computer Science*, Berlin: Springer-Verlag, 1987.

[51] A. D. Janni, "A monitor for complex CAD systems," in *23rd ACM/IEEE Design Automation Conference*, pp. 145–151, 1986.

[52] A. Casotto, A. R. Newton, and A. Sangiovanni-Vincentelli, "Design management based on design traces," in *27th ACM/IEEE Design Automation Conference*, pp. 136–141, 1990.

[53] D. Knapp and A. Parker, "The ADAM design planning engine," in *Artificial Intelligence in Design, Volume II*, pp. 263–285, Academic Press, 1992. reprinted from IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 10, No. 7, July 1991.

[54] D. W. Knapp and A. C. Parker, "A design utility manager: The ADAM planning engine," in *23rd ACM/IEEE Design Automation Conference*, pp. 48–54, 1986.

[55] M. L. Bushnell and S. W. Director, "VLSI CAD tool integration using the Ulysses environement," in *23rd ACM/IEEE Design Automation Conference*, pp. 55–61, 1986.

[56] J. Daniell and S. W. Director, "An object oriented approach to CAD tool control," *IEEE Transactions on Computer-Aided Design*, pp. 698–713, June 1991.

[57] M. F. Jacome and S. W. Director, "Design process management for CAD frameworks," in *29th ACM/IEEE Design Automation Conference*, pp. 500–505, 1992.

[58] T.-C. F. Chiueh and R. H. Katz, "A history model for managing the VLSI design process," in *International Conference on Computer Aided Design*, pp. 358–361, 1990.