MODELING THE CONFIGURATION MANAGEMENT PROCESS

Steven Paul Levi

CU-CS-513-91 January 1991

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

(303) 492-7514
(303) 492-2844 Fax

CONTENTS

FIGURES

CHAPTER 1

INTRODUCTION

As processing capabilities increase in both power and sophistication, information becomes more abstract. A response to a database query may involve multiple views and multiple distributed databases. An executable binary image may be the result of dozens of intermediate processing steps involving a wide variety of tools—compilers, linkers, application generators and library managers. An integrated circuit design may involve hundreds of thousands of circuits and a complex suite of analysis tools to determine its consistency. This thesis describes the Graph Transform (GT) model, an organizing paradigm for managing this kind of complex information. Although capable of being generalized to any computer-aided information manufacturing process, the GT model presented here focuses on the area of software configuration management.

Software configuration management (SCM) is the discipline of controlling the evolution of complex software systems [78]. Today, creating and maintaining evolving software systems is a complex process that has become virtually impossible to consistently manage and control without the use of sophisticated computer programs. Software systems are becoming larger and their useful lifetimes are growing longer. It is not uncommon to find software systems composed of tens of thousands of primitive components existing in many versions and variants which are constructed using very complex manufacturing processes.

In the future, as programming techniques advance, systems that manage programs must become more sophisticated. It is conceivable that few, if any, conventional programs will be written. Instead of writing traditional programs, "Meta-programs" will be composed. These Meta-programs will specify how to configure or combine existing programs into meaningful composite programs. Examples of this can already be seen today—Ala Carte is a system being developed at the University of Colorado at Boulder. Ala Carte configures different database systems out of pre-fabricated parametric components. The selected components are based on the requirements and characteristics of the database specified in a request[19].

In this thesis, the problem of software configuration management is decomposed into three basic concepts: composition, manufacture and consistency. Composition determines all the components, tools, parameters and interconnections that are needed to adequately define a configuration—a complete description of a software system or product. Manufacturing takes the configuration produced in the composition process, constructs a plan that details the most efficient way to build it, and then enacts that plan. Consistency plays an important role throughout the SCM process—how consistency is defined and maintained affects the overall complexity of the process.

These three concepts correspond to three areas of research in software configuration management. The first is model composition[23,71,46,84,67]—the process of consistently determining the set of components that make up a particular version or variant of a software system. The second is complex manufacturing[13,7,42,80,73] which investigates ways to facilitate automating and optimizing complex sequences of complex manufacturing steps. The third is consistency analysis[69,64,36,62] which focuses on analyzing properties of components that comprise a software system.

Until now, these areas were explored in isolation. As a consequence, potential interdependencies that exist between them were not apparent. In this thesis, the GT model provides a unified framework in which all three areas are integrated and their interdependencies are made explicit.

1.1 The Model

The primary contribution of this thesis is the Graph Transform (GT) Model and its ability to describe the software configuration management process. The model, synthesized from ideas and concepts existing piecemeal in other research, focuses on the entire process. Looking at the process as a whole is important given that what makes SCM complex is not only the individual parts, but the interactions between those parts. The GT model is able to:

- Capture the entire SCM process.
- Make explicit inter-dependencies between SCM tasks.
- Define a design space for configuration management systems.
- Show how existing systems are points in that design space.
- Describe more powerful systems within the same design space.
- Provide a unified view of Version Management both for Operators and Components.
- Make explicit the dynamic nature of the SCM process.
- Define and explain how object-specific constraints, selection-specific models, input-sensitive operators and complex predicates interact and affect the SCM process.

The GT model introduces the concept of a configuration and details its consistent composition and manufacture. Explicated as a series of transformations on attributed graphs, the GT model captures the basic SCM process. It makes explicit the interdependencies which necessarily arise among the various SCM sub-processes because of the complexities needed to model real-world problems.

The goal of the GT model is to capture the essence of software configuration management in a technology- language- and database-independent way. The GT model captures and explains the relevant details of existing systems, even though they encompass a diverse spectrum of concerns and functionality. The GT model also provides a vehicle for exploring and researching future systems. Being technology-independent, the GT model suggests logical decompositions that can have many different physical implementations.

As a software system evolves and shifts focus, it is important that the associated configuration management environment also be able to evolve and shift. For example, the system structure—a description of how the components fit together—may change rapidly in the early development of a software system. During this stage resources are needed to dynamically construct the system model. At later stages when the system structure is stable, dynamic construction of the system model is an inefficient use of resources. In the GT model, it is possible to adjust the process to reflect this shift by changing the interpreted program that represents the SCM process.

A configuration management system should also be flexible enough to allow for varying degrees of consistency analysis. At times, there may be a desire to devote resources to full semantic analysis. At other times, especially when the system is known to be inconsistent, simple analysis might be more suitable. The GT model allows the user to change the predicate that is used to determine consistency.

The software configuration management problem is so complex that its solution must be cross-discipline in nature. In order to effectively solve SCM problems, elements from software engineering, databases, programming languages and artificial intelligence are used in the GT model. Software engineering supplies the concepts of abstraction, decomposition and automation. Database concepts such as persistence, queries, and impedance mismatch also play a role. From the programming languages area, there are the concepts of abstract data types, modules, interfaces and implementations. The concepts of domain specific knowledge representations, inference, logic and planning are borrowed from artificial intelligence.

The GT model provides a conceptual understanding of the task being modeled. It is an abstraction mechanism that focuses on important issues, allows for the separation of concerns, and helps clarify potential interrelationships.

1.2 Thesis overview

This dissertation develops by describing the details of the SCM problem and its complexities. After providing sufficient background, the GT model itself is presented. To validate the model, examples of existing systems are described as model instances. Finally, the thesis discusses related work, future work and conclusions. The following is a more detailed overview.

Chapters 2 and 3 introduce the SCM problem. Chapter 2 describes basic software configuration management. It does so by introducing an example and a basic process in which SCM requests are satisfied. The example is then used to walk through the process, step-by-step. Central to this process is the concept of a configuration—a complete specification for a software product. A configuration contains all the components, operators, parameters, derived objects and structural and derivational interconnections needed to build a product. Chapter 3 introduces real-world problems not covered by the basic model. In particular, it introduces the concepts of versions and variants, object-specific constraints, selection-specific models, input-sensitive operators and complex predicates. These concepts are then used to amend the basic model.

Chapters 4 and 5 formalize the software configuration management process described in Chapters 2 and 3 by presenting the Graph Transform model. Chapter 4 distills the information presented in Chapters 2 and 3 into concise, formal knowledge representations. Chapter 5 introduces the Graph Transform model which uses the knowledge representations developed in Chapter 4. The model is explicated as a sequence of transformations performed on attributed graphs. The model describes a transaction that starts with a user's request for a product and terminates when the request is satisfied. The five transformations discussed are Compose, Refine, Merge, Schedule and Build. Throughout the chapter, the complexities introduced in Chapter 3 are re-visited, and the interconnections they induce between the transformations are detailed.

Chapter 6 serves to validate the model. It does so by describing four existing SCM systems Make, DSEE, Odin and Adele as instances of the Graph Transform model. Chapter 7 includes comparisons with other SCM models and other areas of related research, such as software engineering environments and process programming. Finally, in Chapter 8, future research, conclusions and a perspective of this dissertation are given.

CHAPTER 2

BASIC SOFTWARE CONFIGURATION MANAGEMENT

Software configuration management is the discipline of controlling the evolution of complex software products. Broadly defined, a software product is any software object that is the result of the application of a sequence of manufacturing steps to a set of software components. A software product may be as simple as the object produced by applying a compiler to a source program. It may be as complex as an executable binary image that is the result of applying dozens of intermediate manufacturing steps to thousands of source modules. A manufacturing step, also called an operator, is an abstraction mechanism which encapsulates software tools and their behavior.

A configuration, sometimes called a manufacturing graph, represents a plan or blueprint for the construction of a product. It contains all of the components and operators, as well as the structural and derivational interconnections needed to actually construct the product.

Think for a moment about a plan for a car as a complete specification. It contains the specifications for the raw materials and car parts needed to build the car. It may also contain a description of the specific manufacturing steps needed to transform the raw materials into car parts, and car parts into the car itself. The car plan includes structural information about relationships that exist between car parts. This information, similar to that found in an exploded view diagram, details how car parts fit together to form the car. The plan may also contain dependency information. Dependency information constrains the order in which certain steps may be executed. For example, a car part cannot be used in a manufacturing step until the part itself is manufactured.

A configuration for a software product contains information similar to a car plan. The construction of software systems from components is analogous to building a car from parts. The "raw materials" used to build the software system are the primitive components; the "parts" are the intermediate, derived components; and the manufacturing steps are the operators that specify the production of intermediate components and products. Structural information describes how software components fit together to form software systems. Dependency information may impose constraints on the order in which intermediate objects can be constructed.

Software configurations play an important role in SCM. To better understand the importance of this role, a process is developed that describes the construction and execution of software configurations.

The SCM process can be thought of in terms of a transaction. As in database systems, a transaction starts when a user makes a request and terminates when the request is satisfied. For example, a request to a database might be: "Find all the books in the library database that are associated with the 'linguistics' keyword". The result, or satisfaction, of this request is a list of books. In SCM, the request is for a software product and it is satisfied when that product is made available. For example, the request: "Produce a Pascal compiler for a Vax architecture" is satisfied when an executable object that satisfies the request is created. The SCM process that is encapsulated by this transaction is composed of two basic phases: composition and manufacture. In the composition phase the information contained in the user's request is used to build a complete configuration for that product. The manufacturing phase takes the completed configuration and executes it. When the execution of the configuration is completed, the product is available and the transaction terminates.

In order to gain a better understanding of the SCM process, this chapter is devoted to developing and refining the process by way of an example. The example describes the construction of a family of language-machine compilers. Using the analogy of a car plan, the interest is not in the plan (process) for just one car, say for a Honda Accord, but for all Honda cars. The example is discussed in detail because it is referred to throughout Chapter 2 and Chapter 3 and serves to solidify concepts as they are introduced. After the example is presented, the SCM process is developed in greater detail by breaking down the steps involved in the sub-processes of composition and manufacture.

2.1 An Example

In this section, a detailed description of the process for constructing a family of simple compilers is given. Specifically, compilers for the Pascal[41] and Modula-2[87] languages that can run on the Sun4, IBM rs6000, and DEC Vax architectures can be constructed. This example is used throughout the remainder of Chapters 2 and 3 as a means of explicating and motivating the ideas and concepts that are presented.

While basically adhering to well known compiler construction techniques[82], the example takes the liberty of deviating, whenever appropriate, from the standard compiler construction model in order to enhance the example's utility. Compiler construction has been chosen as the proto-typical example for the following reasons:

- Compiler construction is a well understood problem with which most readers are familiar.
- Its decomposition and information flow are well defined.
- Compiler construction decomposition describes a generic model which represents a template for a whole family of language-machine compilers.
- The manufacturing process used to construct a compiler from its components is more complex than a simple compile-and-link sequence.
- In constructing a family of compilers for various language-machine pairs, the re-use of components is obvious.

Figure 1 depicts a simple compiler model and the set of components needed to build the various compilers. In this model, a compiler is composed of three abstractions: a language-dependent front end (FE), a translator (Translator), and a machine-dependent back end (BE), each of which is, in turn, composed of two components.

The lowest level components or leaves of the tree structure shown in the upper

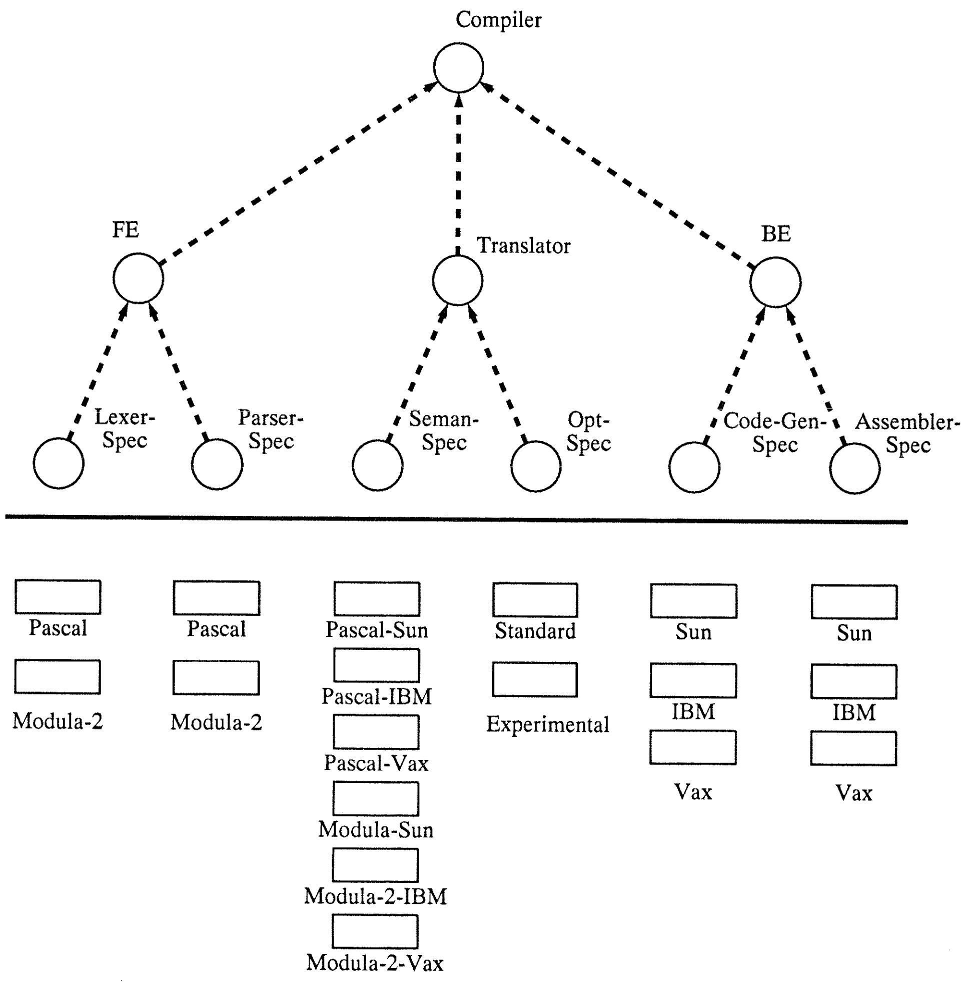

Figure 1: Compiler Construction Components

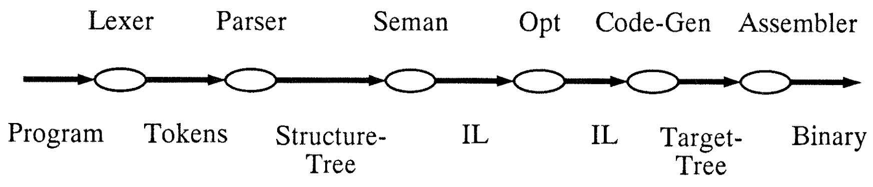

Figure 2: Information Flow in Compiler Construction

half of Figure 1 represents the specifications needed to construct a compiler. At times the specifications will be discussed and at other times their realizations, or what they do, are discussed. For example, Seman-Spec specifies semantic analysis. But its realization performs semantic analysis. When discussing specifications the "-Spec" postfix is used. For example, "Seman-Spec is processed by ..." When talking about purpose, or functionality, the "-Spec" postfix will be dropped. For example, "The purpose of Seman is to ..." Figure 2 shows the information flow between the various realizations of the leaves of the compiler model. It is a pictorial representation of the flow of information described below.

The FE of a compiler is language-dependent. It uses textual programs written in a specific programming language as input and performs language-dependent structural analysis. The FE produces a stylized representation of the original input. The FE itself is composed of two abstractions representing the tasks of lexical and syntactic analysis. In Figure 1, the lexical analysis abstraction, "Lexer" is represented by the realization of the "Lexer-Spec" object. Lexer uses as input textual programs written in a specific language, and produces a stream of tokens for that language. The syntactic analysis abstraction, "Parser", is represented by the realization of the "Parser-Spec" object. Parser uses the stream of tokens produced by the Lexer as input, checks for valid sentences in the language and builds up a stylized representation.

The Translator uses the structure produced by the FE as input and transforms it into a structured, machine-independent representation. The Translator abstraction itself is composed of two abstractions: semantic analysis ("Seman") and machine-independent optimization ("Opt"). Seman, a realization of "Seman-Spec", is both machine and language dependent. Its purpose is to discover, using the semantics of the language, the meaning of the original program and translate it into a machine-independent intermediate-language representation (IL). While not exactly consistent with standard compiler construction techniques, assume that the Seman abstraction also performs some machine-dependent computations. This introduces at least one component that is dependent on both the machine and the language (to make the SCM aspects of this example more interesting). The "Opt" abstraction, a realization of "Opt-Spec", takes the IL structure produced in Seman and performs machine-independent optimizing transformations on it. The result is a optimized version of the IL representation. Opt is introduced into the model so that at least one component is not dependent on either the language or the machine.

The BE of a compiler is machine dependent. It uses the machine and language independent structure constructed by the Translator as input and produces an object that is capable of being executed on the targeted machine. The BE abstraction is composed of the code generation ("Code-Gen") and assembly ("Assembler") abstractions. The Code-Gen object, a realization of "Code-Gen-Spec", determines execution order and performs resource allocation. The Assembler, a realization of "Assembler-Spec", performs address resolution and operator encoding and produces a machine interpretable object.

In a production level compiler, the abstractions described in this simple model would be further refined and the set of actual components needed would be considerably larger. For the purposes of exemplifying aspects of configuration management, this level of decomposition is more than adequate. In fact, more detail would obfuscate the details of the SCM process.

Compiler Specifications

In order to specify the compilers described in this example, three specification techniques are used: one for lexical analysis specifications (Lexer-Specs), another for syntactic analysis specifications (Parser-Specs) and a third to specify the rest of the compiler. Lexer-Specs are regular-expressions and describe the lexical tokens emitted by the targeted source language. They serve as input to lexer-generator tools. Parser-Specs are context-free grammars. They describe the syntactic structure of the targeted source language. Parser-Specs serve as input to Parser-generator tools. The remainder of the compiler is specified in the Modula-2 programming language.

Operators for Compiler Construction

Each of the specification techniques described above requires a different operator, or sequence of operators, to transform the specification into an object or objects the machine can understand. Operators specify the input and output behavior of tools (or families of tools). Figure 3 shows the various operators that are needed to carry out the compiler construction process as defined in this example. They include a Lexer-generator, a Parser-generator, two compilers and a linker. The remainder of this section explains the operational characteristics of these operators.

Lex-Gen represents an abstract operator for lexical-generator tools. The Lex-Gen operator uses a lexical analysis specification, in the form of a regular-expression grammar (Lexer-Spec), as input and produces two objects: "Lex-Gen-C-Source" and "Lex-Gen-Tokens". Lex-Gen-C-Source is a C-Source file. When compiled, the resulting executable object is a Lexer. This Lexer accepts a programming language text stream as input and produces a sequence of tokens that conforms to the original specification and can be understood by the Parser. The Lex-Gen-Tokens object specifies the tokens emitted by the Lexer and defines the Parser's input stream.

Parse-Gen represents an abstract operator for parser-generator tools. Parse-Gen uses two objects as input a Lex-Gen-Tokens object, produced by the Lex-Gen operator,

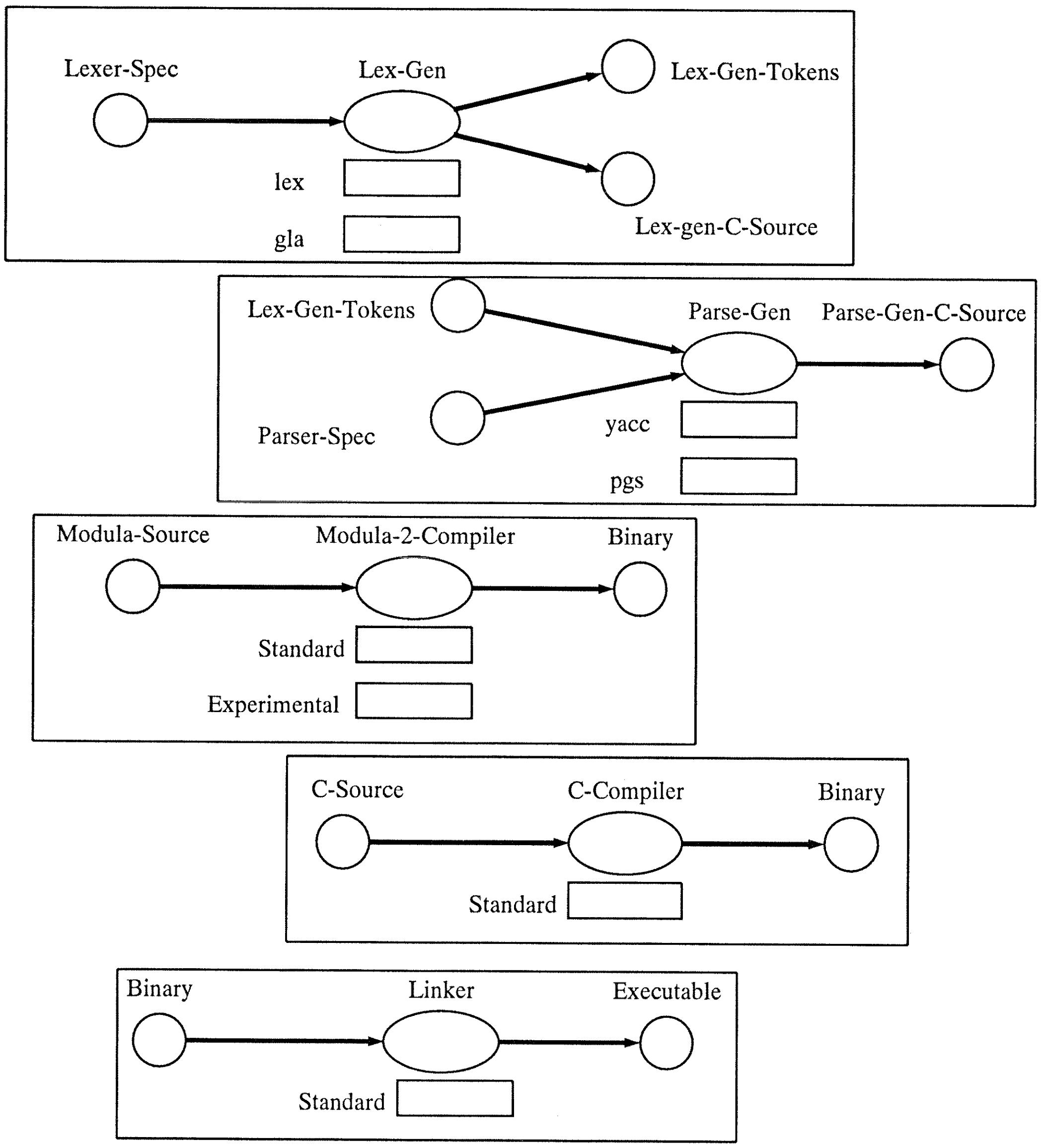

Figure 3: Operators For Compiler Construction

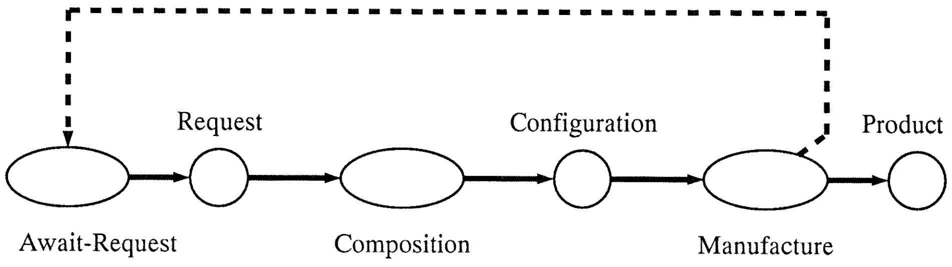

Figure 4: The SCM process

and a Parse-Gen-Spec object , a parser specification in the form of a context free grammar. Using these two inputs, the Parse-Gen operator produces a Parse-Gen-C-Source object. Compilation of the Parse-Gen-C-Source results in a Parser. The Parser accepts a token stream constructed by the Lexer, checks for valid phrases in the language and produces a stylized representation of the original program.

The Modula-2-Compiler operator takes as input a Modula-2 source file and converts it into a binary image. The C-Compiler operator takes as input a source file for the C language and converts it into a binary object. The Linker operator takes as input a set of binary objects, links them together and produces an executable object.

2.2 The SCM process

Figure 4 depicts an abstract view of the SCM process. The circles represent objects and the ovals represent processes. The overall process is a loop that waits for requests and then processes them. What occurs between the time when a product request is received and the time when the product is made available is important. What occurs during the elapsed time is called a SCM transaction.

A SCM transaction starts with a request for a product. A request contains a component, a product and a set of initial constraints. The component represents the root object from which the product is to be derived. The product represents the type of object to be derived from the component. The initial constraints define certain properties that must hold for the product.

Once a request is made, it is fed as input to the composition process. The composition process takes the request and builds up a complete configuration for the product. The configuration is then fed as input to a manufacturing process that executes, or manufactures, the product described by the configuration. When the configuration execution is completed, the product is made available and the process or transaction terminates.

In the rest of this chapter, the process within the SCM transaction is developed

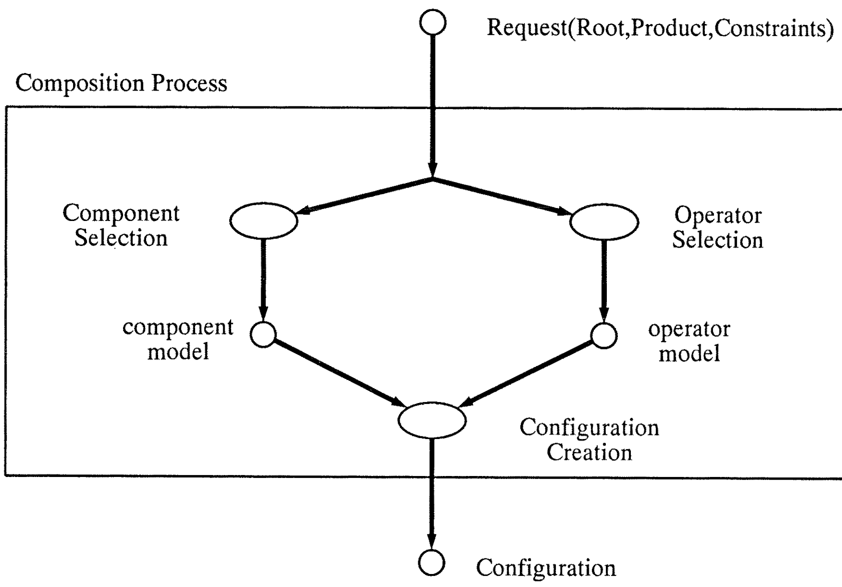

Figure 5: The SCM Composition process

in greater detail by using an example of an actual request and then walking through the transaction, step-by-step. The request to be satisfied is: "Produce an executable Pascal compiler that will run on a Vax architecture" (Pascal-Vax-Compiler). For this request, the root component is a Compiler object, the product is "executable" and the initial constraints are "language=Pascal" and "machine=Vax".

2.2.1 Composition

Once a request exists, it must be built up into a complete configuration. To do so the following information must be inferred:

(1) Find/select all the components and the structural interconnections needed to construct the product.
(2) Find/select all the appropriate operators and the derivational interconnections needed to transform the components into the product.
(3) The merging and expansion of the information found in (1) and (2) which produces a complete configuration.

The composition process depicted in Figure 5 is composed of three sub-processes: component selection, operator selection and configuration construction. The rest of this section is broken into three sub-sections, one for each sub-process. In each sub-section, a simple overview of the necessary steps is given and is then related to the satisfaction of the Pascal-Vax-Compiler request.

2.2.1.1 Component Selection

The process of determining the components needed for a complete configuration is composed of two sequential sub-processes: generic component selection and specific component selection. The goal of generic component selection is to find all the generic components and the structural interconnections that are needed to construct a generic product. Specific component selection is a refining process. Its goal is to take the structure produced in generic component selection and substitute the generic components with the appropriate concrete components needed to satisfy the actual request.

Components are objects that can be either abstract or concrete. Abstract components and their interconnections are a structuring mechanism. Abstract components also serve as generic components, or place holders. Looking at Figure 1 on page 7, all of the abstract and concrete components needed to build the entire family of compilers are found. The components above the solid line, denoted as circles, are abstract. These abstract components, together with their arrows, describe the abstract structure or "generic model" of a compiler. The components of this model not only describe the structure but also serve as generic place holders. The components denoted as rectangles below the solid line are concrete components. Concrete components represent refinements or instances of the corresponding generic components depicted directly above them.

To satisfy the Pascal-Vax-Compiler request, the generic model needed to construct a generic compiler is constructed. The generic model is then refined and tailored specifically to a Pascal-Vax compiler.

A Generic Component Model (gCM) describes the generic components and their structural interconnections needed to build a product. As stated earlier, the circles and arrows above the solid line in Figure 1 on page 7, represent a generic component model for constructing a generic compiler. It says that all compilers are composed of an FE, a Translator, and a BE. These in turn are composed of a Lexer-Spec and a Parse-Spec; a Seman-Spec and an Opt-Spec; and a Code-Gen-Spec and an Assembler-Spec, respectively. For and compiler request, this structure describes the gCM needed.

Generic component models are useful abstraction mechanisms. They allow information about a family of systems to be centralized in one place. This generic structure is the same for every language-machine compiler pair that can be constructed in the model. Even those who are unfamiliar with the general structure of a compiler can look at the gCM for a compiler and glean its basic structure.

Real compilers translate textual programs written in a specific language into objects that can be executed on a specific hardware architecture. For this transaction the Pascal language and the Vax architecture are used.

In order to build such a compiler, the generic components in the gCM are replaced with suitable concrete components. A generic component model in which the generic objects have been replaced with an appropriate and consistent set of concrete components is said to be a Specific Component Model (sCM)[1].

[1]sCM (specific Component Model) is not to be confused with SCM (Software Configuration Management)

For this specific request, the generic component model for a compiler is refined by selecting those concrete components that are appropriate for a Pascal-Vax-Compiler. This is done by selecting one concrete component from the set of concrete components found below the solid line in Figure 1 for each leaf in the model. The remaining paragraphs in this section discuss the reasoning behind each specific substitution. Although a process exists for performing these substitutions, its description is deferred to a later section. What is described here is the reasoning behind the selection choices.

Because the FE is language dependent, the concrete components for the Pascal language must be selected. As a consequence, the generic Lexer-Spec component is replaced with the concrete Pascal-Lexer-Spec component and the generic Parser-Spec component is replaced with the specific Pascal-Parser-Spec component.

Because the BE is machine dependent, the concrete components for the Vax architecture must be selected. Therefore, the generic Code-Gen-Spec component is replaced with the concrete Vax-Code-Gen-Spec component and the generic Assembler-Spec component is replaced with the concrete Vax-Assembler-Spec component.

The Translator is an interesting case. One of its components, Seman, is both language and machine dependent. In contrast, its other component, Opt, is independent of both language and machine. Because the Seman-Spec is both machine and language dependent, a concrete component that handles both of these dependencies must be found. To satisfy the Pascal-Vax-Compiler request the generic Seman-Spec component is replaced by the concrete Pascal-Vax-Seman-Spec component. For the Opt-Spec generic component, there are two concrete instances: Standard-Opt-Spec and Experimental-Opt-Spec. The request for a Pascal-Vax compiler does not distinguish between standard and experimental components. There are now two basic choices. Either to abort the transaction and inform the user that the initial request must include a preference for either the standard or experimental components. Or, to supply a default selection mechanism. In this model, a default selection mechanism is the preferred solution. For now, assume that the default mechanism selects the first concrete component in the list. Therefore, the Opt-Spec generic component is replaced with the Standard-Opt-Spec component[2].

At this point all of the necessary substitutions have been performed and the specific component model for a Pascal-Vax compiler has been constructed.

2.2.1.2 Operator Selection

Like component selection, the process of performing operator selection is composed of two sequential sub-processes: generic operator selection and specific operator selection. The goal of generic operator selection is to find all the generic operators and the derivational interconnections that are needed to construct the product. Specific operator selection, like specific component selection, is a refining process. Its goal is to take the structure produced in generic operator selection and substitute the appropriate

[2]Note that had a request for a compiler been made without clarifying language or machine, then by the rule of default selection, a Pascal-Sun compiler would have been produced. This can be seen by inspecting the first element in the rows of concrete components found in Figure 1 page 7.

generic operators with the specific concrete operators needed to satisfy the particulars of the request.

Operators, like components, are also objects. As stated earlier, they can be viewed as abstractions that encapsulate tools and tool behavior. Operators capture the derivational relationship that exists between two sets of components—both its input set and its output set. Like components, operators can be either abstract or concrete. Figure 3 depicts the operators needed to satisfy the Pascal-Vax-Compiler request. The circles represent components, the ovals represent generic operators and the rectangles represent concrete operators.

To satisfy the Pascal-Vax-Compiler request, it is necessary to determine the generic operators and structure needed to construct a compiler. The generic model is then refined according to the information found in the request.

A Generic Operator Model (gOM) describes the generic operators and their derivational interconnections needed to build a product. Figure 6 depicts the gOM needed to derive a compiler. The gOM in Figure 6 can be constructed using the request and the operators found in Figure 3. The gOM is like a completed jigsaw puzzle in which the operators are like jigsaw pieces. These pieces have irregular shapes and fit together in a complex way. When looking at the completed puzzle, it is clear how the pieces fit together. But when looking at the each piece in isolation their interconnections are not immediately apparent.

Like the process of constructing a jigsaw puzzle, the process of constructing a gOM is quite complex, especially as the number of operators that the system supports becomes large. This process, discussed in detail in Section 5.3.4, involves type inferencing over derivational relations. These relations are contained in an object called an Operator Graph, and are discussed in Section 4.2.5. For now, assume that the process exists and that the gOM in Figure 6 can be constructed from the request and information about operators.

The gOM in Figure 6 is similar in structure to the generic component model for compilers found in Figure 1. They differ only in the semantic interpretation of the vertices and edges. The gOM for constructing a compiler can be interpreted as follows:

- A compiler is composed of three kinds of specifications.
- From each of these specifications there exists a derivation sequence–a sequence of one or more operators that results in the production of binary images.
- Finally, these binary images can be combined by the application of an operator to produce an executable image.

This gOM is applicable to the construction of all compilers described in the example.

A Specific Operator Model (sOM) is defined as a gOM in which all generic operators have been replaced with appropriate and consistent concrete operators. The process of transforming a gOM into a sOM is identical to the process of transforming the generic component model into the specific component model described in Section 2.2.1.2.

If you refer back to Figure 3, you will see that the rectangles denote the concrete operators available. In the specific request for a Pascal-Vax compiler, there is no mention

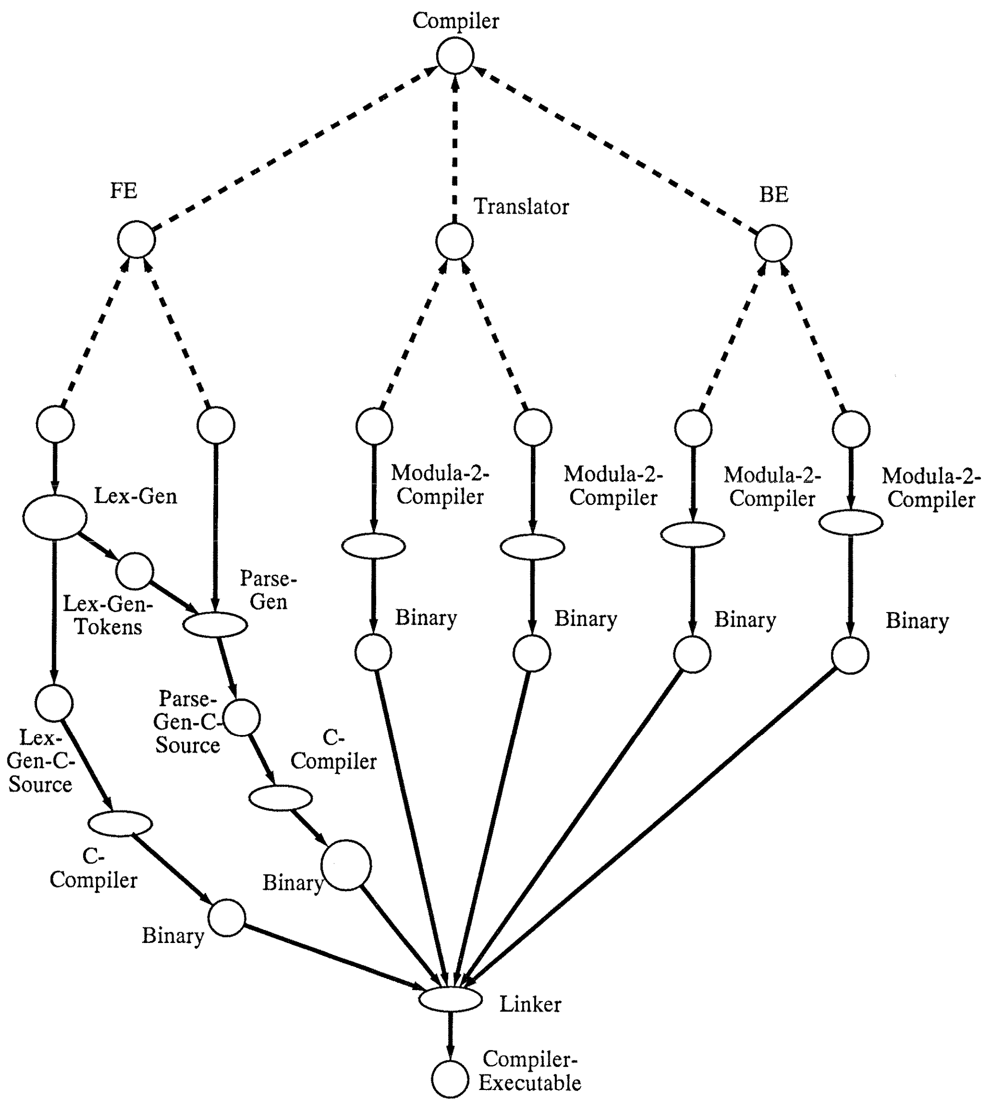

Figure 6: The gOM for compiler construction example

about the choice of operators. Therefore, when performing the substitution, the default selection rule of picking the first concrete operator in the list must be used. Using this rule, the following substitutions are made:

- Lex-Gen → lex-Lex-Gen
- Parse-Gen → yacc-Parse-Gen
- Modula-2-Compiler → Standard-Modula-2-Compiler
- C-Compiler → Standard-C-Compiler
- Linker → Standard-Linker

2.2.1.3 Complete Configurations

At this point both the specific component model and the specific operator model have been constructed. The two must be merged and the missing pieces must be filled in, forming a complete configuration. Merging the two models into a configuration, like the process of producing the gOM from the operator set, is a complex task that is described in detail in Section 5.3.5. For now it is assumed that the process exists and that Figure 7 is produced by that process.

Figure 7 represents a complete configuration for the Pascal-Vax compiler and represents a plan for its manufacture. It contains all the concrete components, operators, and structural and derivational interconnections needed to completely specify a Pascal-Vax compiler. With the configuration complete the transaction is ready to carry out the manufacturing plan.

2.2.2 Manufacture

Software manufacturing is similar to car manufacturing. In car manufacturing, the plan or blueprint for a car defines a partial specification for the construction of the car. Once a plan exists it is necessary to completely specify the construction of the car. An assembly line is designed and constructed to manufacture the car. The assembly line design takes into consideration the order in which each manufacturing step must be performed and is structured to perform tasks in parallel whenever possible. The primary consideration in the assembly line design is to minimize the time it takes to go through the assembly line without sacrificing the integrity of the car's construction. When the assembly line goes into production, most of the manufacturing process is automated. There are few, if any, decisions left to be made.

In software manufacturing, the configuration is the plan for constructing a software product. Like a car plan the configuration partially specifies the construction of the software product. To completely specify the construction of the software product, an operator schedule, similar to an assembly line, is constructed. As in manufacturing a car, the primary consideration in designing the operator schedule is to minimize the time it takes to produce the product without sacrificing the product's integrity. When the operator schedule is put into production, the process of manufacturing the product is fully automated.

The configuration for the product specifies what the product is and to some

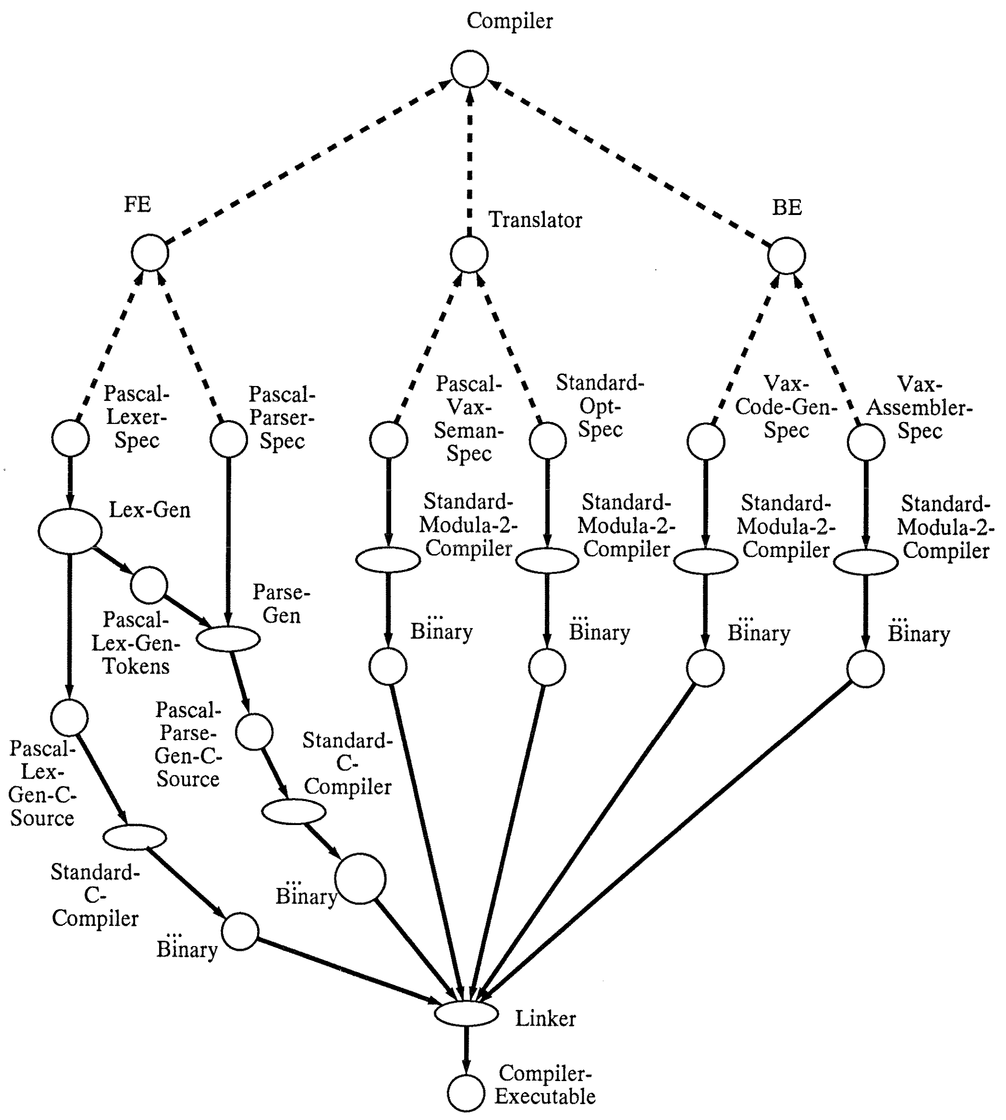

Figure 7: The complete configuration for the compiler construction example

extent how it will be constructed. It describes the objects and interconnections needed for software manufacture. The kinds of objects found in a configuration include a set of primitive concrete components, a set of concrete operators, the intermediate derived components produced by the application of operators, and the actual product.

The configuration for a Pascal-Vax-Compiler specifies the operators and the components needed to construct such a compiler. It also contains the intermediate components that are byproducts of operator application. An example of a byproduct in this configuration is Pascal-Lex-Gen-Tokens, produced by the lex-Lex-Gen operator.

In addition, a configuration contains interconnections between objects in the form of relations. This information is important for manufacture. As described in section 2.2.1.2, operators capture the derivational relation between its input set and its output set. This dependency relation is transitive[3]. For example, in the Pascal-Vax-Compiler configuration depicted in Figure 7, the output of the yacc-Parse-Gen operator depends on two inputs, one of which depends on the output of the lex-Lex-Gen operator which, in turn, depends on its input. This relation also defines a partial ordering over the objects which constrains the execution order of the operators described in the configuration. One constraint is that the yacc-Parse-Gen operator cannot execute until the lex-Lex-Gen operator produces the Lex-Gen-Tokens object needed as input.

In summation, configurations and the derivational dependency relation they contain are used extensively in manufacturing to determine what operators to apply in what order to produce the product. In the next section, the basic manufacturing process is described and the notion of an operator schedule is developed. The following section elaborates on the process by describing how certain components can be re-used. Next, change is introduced into the process and its impact and implications for manufacture are described. Finally, the notion of overlapping configurations where components are shared in more than one configuration is presented.

2.2.2.1 Basic Manufacturing

The manufacturing process performs the construction of the product specified in a configuration. At the most elementary level the manufacturing process is quite simple because virtually all the necessary information is contained in the configuration. Figure 8 depicts an overview of the manufacturing process which is composed of two sub-processes: schedule and build. The process starts with a configuration that is then transformed into an operator schedule which, when executed, produces the product.

The schedule process uses a configuration as input and produces an ordered list of operators to be executed. As with the other complicated processes, the details for constructing the operator schedule is discussed later in Section 5.4.1. For now, assume that such a process exists.

Figure 9 shows a simplified version of the operator schedule constructed from the Pascal-Vax-compiler configuration by the schedule process. The first column shows the input to the operator and the second shows the concrete operator that are applied.

[3]If A $\rightarrow$B and B$\rightarrow$C then it can be transitively inferred that A$\rightarrow$C.

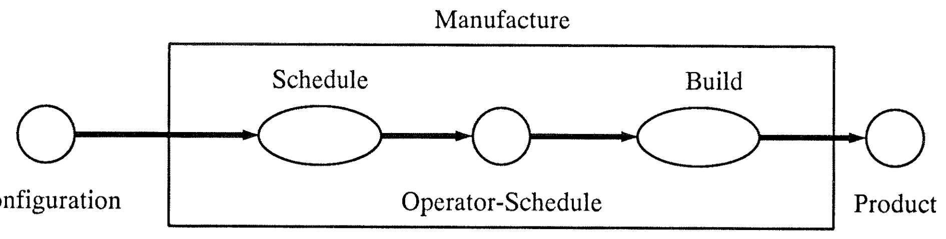

Figure 8: The SCM manufacturing process

The build process uses the operator schedule as input and produces the product by executing the actions, or manufacturing steps, described therein. When all the operators on the list have been executed, the product is made available and the transaction terminates.

2.2.2.2 Reuse

Manufacturing is expensive. Performing configuration manufacture for certain software products may take hours or even days. As a consequence, it is important that no unnecessary manufacturing steps be performed. Unlike manufacturing cars, in SCM the same request may be issued over and over for a given product. Intermediate "parts" produced by a previous request may already exist. As a consequence, when manufacturing software it is possible to short-circuit part of the build process. This is accomplished by checking the "consistency" of an operator with respect to its inputs and outputs. If, at the time the transaction starts, it is known that there exists a set of consistent operators then those operators need not be re-executed.

The notion of consistency plays an important part in the entire SCM process and is discussed in detail in Chapter 5. A simple notion of consistency is introduced here to describe the concept of reuse. In order to construct a simple definition for the term "consistent" it is necessary to first introduce the concept of stateful objects. This is done by associating objects with other properties beyond just their value. One such property, "Created", describes the time at which the value of the object was created.

As stated earlier, an operator establishes a relation between its input and its output. An operator is said to be consistent if the application of a predicate to the operator yields true. An example of a predicate is time-stamp. Simply stated, the time-stamp predicate yields true if the values of the Created properties for all the output objects of an operator are greater than the values of the Created properties of all of the operator's inputs (i.e., if the output is "newer" than the input). An operator is said to

INPUT	OPERATOR
Pascal-Lex–Spec	lex-Lex-Gen
Pascal-Parse-Spec	yacc-Parse-Gen
Pascal-Vax-Seman-Spec	Standard-Modula-2-Compiler
Standard-Opt-Spec	Standard-Modula-2-Compiler
Vax-Code-Gen-Spec	Standard-Modula-2-Compiler
Vax-Assembler-Spec	Standard-Modula-2-Compiler
Pascal-lex-Lex-Gen-C-Source	Standard-C-Compiler
Pascal-yacc-Parse-Gen-C-Source	Standard-C-Compiler
{set of binaries}	Standard-Linker

Figure 9: Operator Schedule for Pascal-Vax-Compiler

be inconsistent if the application of a predicate yields false. Any predicate applied to an operator whose output value is undefined will always yield false—if the objects produced by an operator do not exist then their Created property values are undefined and the predicate is meaningless.

Now that the concept of consistency has been defined, it is possible to make a request to the system. Assume that the time-stamp predicate is used to determine consistency and that no other transactions have taken place (i.e., the system is starting with a clean slate). To satisfy the request, the composition process constructs a configuration and the manufacturing phase then produces the product. Because not one of the operators in the configuration has ever been executed, none of their output objects exists. As a consequence, the application of the time-stamp predicate, or any predicate for that matter, to the operators in the configuration yields false, or inconsistent. This is the same situation as described above in the basic manufacturing process. To satisfy the request, a complete operator schedule is constructed and then built. As each operator is executed, its output objects are created and their Created property is set to their creation time. When the build process is completed, the product has been manufactured and the transaction terminates.

With no change to any object values or their Created properties, the same request is issued again. This time when the time-stamp predicate is applied to the operators in the configuration, the predicate yields true, or consistent, for all of them. Since all operators in the configuration are consistent, the schedule process produces an empty operator schedule. As a consequence, the build process has no work to do and the transaction terminates.

2.2.2.3 Change

A fundamental difference between software manufacturing and car manufacturing is their concept of change. In car manufacturing, once the plan and the assembly line are designed and constructed very little change takes place. A car manufactured today will typically be made with the exact same process and with exactly the same components as a car made six months from today. In software manufacture, change is pervasive.

To talk about change a mechanism that produces change must exist. To this end, a Change transaction is introduced. A Change transaction uses an object as input, performs some action, and produces a modification in the state of that object. For example, if the Lexer-Spec object is submitted to a Change transaction in which its value is edited, then the value and Created properties for that object are modified to reflect the change.

In order to aid the operator scheduler in making computations, the Change transaction broadcasts to all of the operators that transitively depend on the object being changed, that they are no longer consistent. This is possible due to the transitive nature of the dependency relation. To produce and use this broadcast information it is necessary for operators to be stateful. To this end, associated with every instance of a concrete operator is a "Consistency" property. The Consistency property can take on one of three values: "consistent", "inconsistent" or "undefined". The new definition of operator consistency is then enhanced to include these values. It states that if the value of an operator's Consistency property is:

(1) "consistent", then the operator is said to be consistent.
(2) "inconsistent", then the operator is said to be inconsistent.
(3) "undefined", then operator consistency is determined by the application of a predicate, such as the time-stamp predicate given above.

Assume that a request for a Pascal-Vax-Compiler has been made and that the request was satisfied, i.e., the compiler was constructed. The new compiler was tested, a bug was found, and a Change transaction, was enacted on the Pascal-Lexer-Spec. As a consequence of this transaction the broadcast mechanism changed the Consistency property of the lex-Lex-Gen, yacc-Parse-Gen and the Standard-Linker operators by setting their values to "inconsistent". Now a request is again made to the system for a Pascal-Vax-Compiler. This time as the schedule process applies the new definition of consistency, it places only the three operators mentioned above in the operator schedule. To construct the product, the build process only applies the three operators found in the operator schedule, and changes their Consistency properties to "consistent".

2.2.2.4 Overlapping Configurations

So far, the construction and maintenance of only one of the compilers has been discussed in isolation. As stated earlier, the environment described supports the construction of an entire family of compilers. One of the reasons that this example is used is that several of the concrete components that comprise a specific compiler reside in more

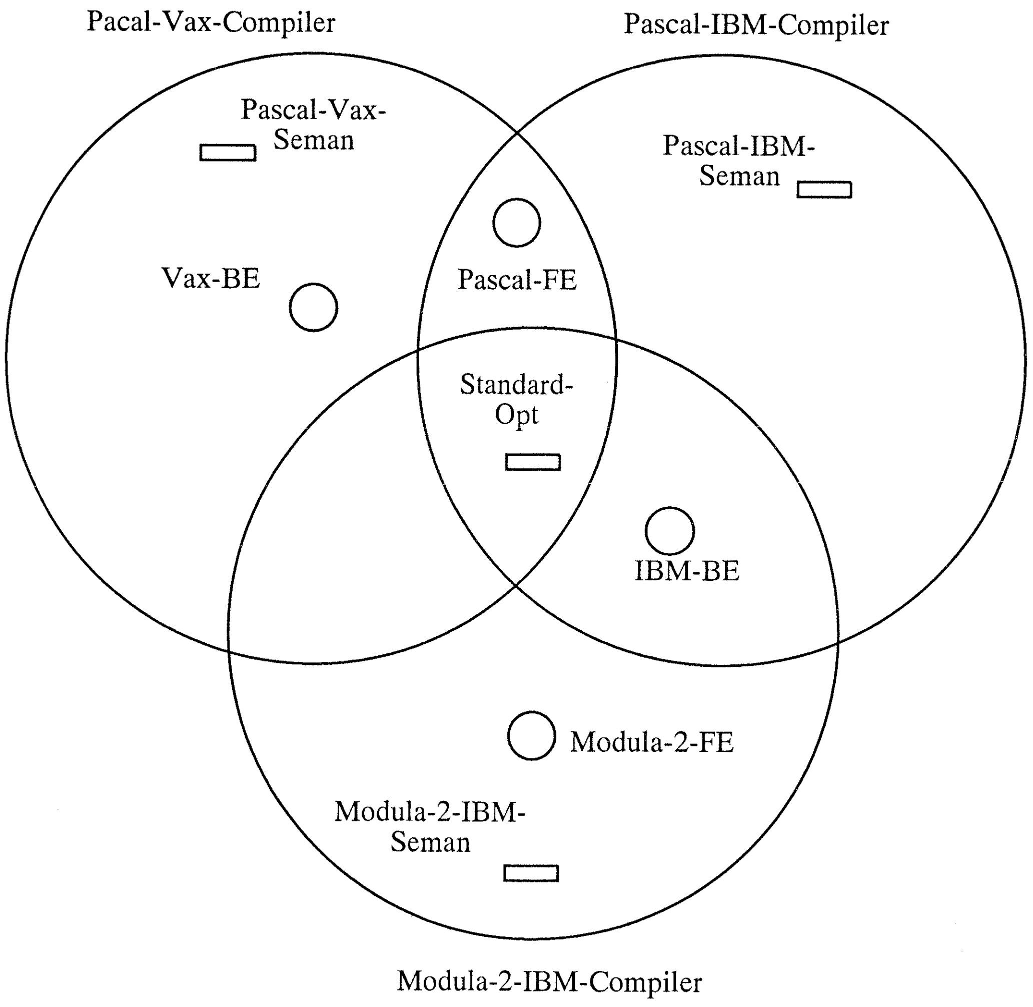

Figure 10: Overlapping Compiler Configurations

than one configuration.

Figure 10 depicts three configurations and how their components overlap. The IBM back end (IBM-BE) is shared by both the Pascal-IBM-Compiler and the Modula-2-IBM-Compiler configurations. Similarly, the Pascal front end (Pascal-FE) is shared by both the Pascal-IBM-Compiler and and the Pascal-Vax-Compiler. The concrete Standard-Opt-Spec component is part of all three configurations.

Assume that there is a clean system, and that a request has been made for a Pascal-IBM-Compiler. To satisfy this request, all of the operators in the resultant configuration are included in the operator schedule and are executed. If the next request is for the Modula-2-IBM-Compiler, then those operators needed to construct the IBM-BE are already in a consistent state and need not be re-executed. The operator that transformed the Standard-Opt-Spec into a binary does not need to be re-executed either. Additionally, if the next request is for a Pascal-Vax-Compiler, then the Pascal-FE and Standard-Opt-Spec transformer do not need to be re-executed.

At this point, if a Change transaction is enacted on the Standard-Opt-Spec component, all three configurations are affected. For the broadcast mechanism to work, it now must have access to the derivation relations for not just one but all three configurations. Having enacted the Change transaction, a request is made for one of the three compilers. The operator for transforming the Standard-Opt-Spec and the Standard-Linker is put into the operator schedule and then executed. As a consequence of this request, the Consistency property of the operator for transforming the Standard-Opt-Spec is set to "consistent". Now, if a request is made for either of the other two configurations, only the Standard-Linker operator for that particular configuration has to be re-executed to produce the product.

CHAPTER 3

COMPLEXITIES IN THE SCM PROCESS

In the previous chapter, a basic model for the software configuration management process was described. In this chapter complexities that perturb that basic model are introduced. These perturbations come about because of the complexities that are inherent in developing and maintaining large software systems in which components change, operators change and interconnections change. Even the process by which this software is managed is susceptible to change.

Each time an object is changed, a new copy or version of that object is created. As a consequence, there is an explosion in the number of objects managed by a system. In order to manage the versions and variants created by change, the concept of an object family is introduced in Section 3.1.

In Section 3.2 , the discussion proceeds to operator parameters. Operator parameters affect the behavior of operators and the derived objects produced by the application of those operators. In order to distinguish between derived objects whose values differ only as the consequence of different operator parameter values (i.e., they use the same concrete operator instance and have the same input) the concept of a derived object family is introduced.

Complexities that can be introduced in the composition process are discussed in Section 3.3. This section is composed of three subsections, one for each of the principle transformations used in the composition process: Compose, Refine and Merge. Section 3.4 discusses potential complexities that can be introduced into the manufacturing process. It is divided into two subsections, one for the Schedule transformation and one for the Build transformation.

3.1 Object Families

In the previous chapter, an association was made between generic objects and a set of related concrete objects. In this section, the notion is elaborated on by describing concrete objects in terms of the attributes associated with them. Objects with attributes in common are then used to introduce the concept of an object family. Object families are important because they capture both the concept of variants (instantiations) of generic objects and the conventional concepts of versions and revisions that come about as the consequence of change.

In the previous chapter, specific models were created from generic models by using a refinement process. Each generic component in the model was replaced with a unique concrete component. The selected concrete component was one of a set of similar components and was chosen because it satisfied the requirements imposed by the initial request. The members of a set of similar components associated with a generic object are

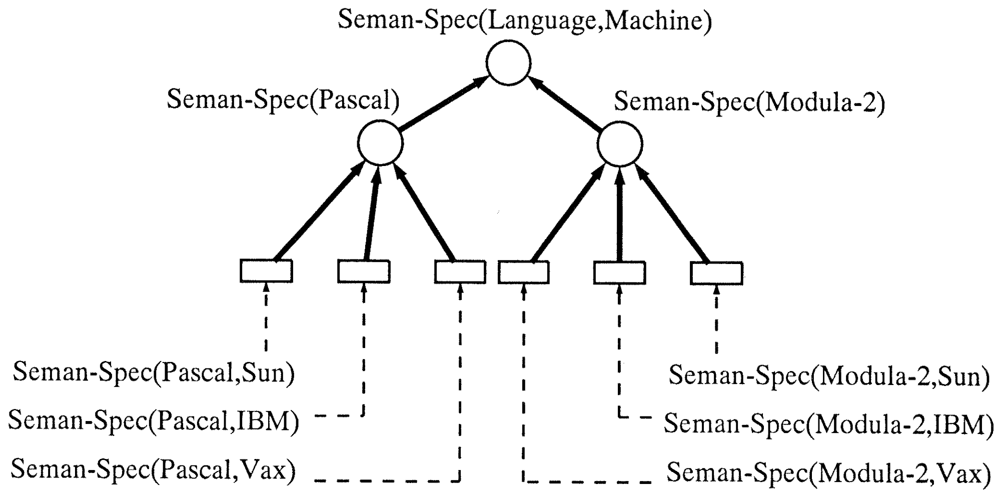

Figure 11: The Seman-Spec Component Family

called the variants or instances of a generic object. Together, the generic object and its associated variants are called an object family. Component, operator and derived object families are all specializations of the object family concept.

In Figure 1 on page 7, a generic model for a compiler was shown. In this model one can find a generic component called Seman-Spec. The concrete components directly below this generic component and below the solid line depict the variants, or alternative implementations, of the Seman-Spec generic component.

This set of components is structured. A family of objects is defined as a set of objects that have the same number and kind of attributes. Figure 11 shows a more refined depiction of the Seman-Spec component family. The root of this family is the Seman-Spec generic component. It defines two attributes "language" and "machine". Below the root are various refinements or members of this family. At the next level down are two sub-families, one for all of the members that have "Pascal" for the value of their language attribute, and one for all the members that have "Modula-2" for the value of their language attribute. At the next level down, because there are only two attributes defined for this family, there are concrete components. These are components in which all of the attributes defined in the root generic component have specific values. An Object family can have an arbitrary fixed number of levels depending on the number of attributes defined in the generic object at its root.

For simplicity, the attributes are ordered. In the Seman-Spec example above this means that the value of the language attribute must be instantiated before the value of the machine attribute. This corresponds to Wiebe's concept of partial instantiation[84]. One can also view families in terms of Prolog unification[15]. If a component is represented as a n-ary fact (where the arity of the fact equals the number of attributes defined for the family), then the all of the objects returned from a Prolog query, in which all the literals were uninstantiated (or "dont-care") would be elements of the family (e.g., `Seman-Spec(_,_)`). The next sub-family would comprise all of the elements returned from a Prolog query in which the first literal was instantiated and the rest were not (e.g. `Seman-Spec(pascal,_)`). Sub-family refinement continues until all literals are instantiated.

Software objects evolve over time. Source code is modified, paragraphs in a chapter are re-arranged, bugs must be fixed, new instances of tools are introduced, or a new capability may be added to an existing program. As software objects evolve over time, new instances of those objects must be created and maintained. In conventional source control systems such as RCS[79] and SCCS[56] specific concepts for versions and revisions of source objects are defined. Versions represent alternative lines of descent and revisions represent sequential changes made to specific versions. A general problem associated with such systems is that they impose a strict definition for these terms. Today, characterizing what constitutes a version of a software object is less well defined[77,51,66]. At an intuitive level, two objects are said to be versions of one another if they share something (enough) in common[81]

To overcome the strict definitions found in conventional source control systems and to extend the concept of versions and variants to apply to all manner of objects, the concept of object families is expanded to include the more conventional concepts of version and revision. This is done by simply associating more attributes with generic objects. For example, if "version" and "revision" attributes were added to the Seman-Spec component family then the family tree would be two levels deeper. The "version" attribute could still distinguish between alternate implementations, or lines of descent, and the "revision" attribute could still describe sequential modifications made to specific versions.

3.2 Operator Parameters

In the previous chapter, operator behavior was specified exclusively in terms of its input and its output characteristics. This is not sufficient for capturing the behavior of most operators. Many operators have parameters or options that modify their behavior. For example, the "debug" option for a compiler operator might modify the behavior of that operator in such a way that its binary output would include symbolic debugging tables. Another, more complicated operator specifies the behavior of the GAG system[3]. GAG is a system for processing attribute grammars. One of the more interesting features of the GAG system is the number of options it supports. The number and the variations of possible options is so large that the system supplies a context-free grammar for describing them. The following is just a few of the names of the non-terminal productions in the option grammar: $ANALYZE, $DEPENDENCY, $OPTIM, $GENERATE, $TRANSDEF, $VSTRANS, $PARSER, $SCANNER, $EXPAND, $ORDER, and $ARRANGE. All of these options are important because in some way, they affect the processing of GAG's input and therefore have an effect on the objects produced.

Just as attributes associated with generic objects produce object families, parameters associated with operators produce derived-object families. Derived-object families are important because they provide a mechanism for describing, selecting and maintaining multiple versions of derived-objects. In some conventional SCM tools such as Make[28], parameters are not used to distinguish derived-objects. A consequence of this is that it is not possible to support simultaneously "debugged", "standard" and "optimized" versions of a system. Because no distinction is made between these objects, they in effect become interchangeable, which is undesirable in many circumstances.

3.3 The Composition Process

In the last chapter, the composition process was depicted as a well-structured, well-ordered sequence of steps that used the Compose, Refine and Merge transformations. A generic component model was composed (Compose transformation) and then refined (Refine transformation). A generic operator model was composed (Compose transformation) and then refined (Refine transformation). Finally, the two resulting specific models were combined (Merge transformation) to form a complete configuration.

In this section, potential complexities that arise from real-world problems are introduced. They not only potentially perturb the transformations but can have an impact on their sequencing. Three sub-sections address these real-world problems, one for each of the three transformations mentioned above. The order of Refine and Compose is reversed due to their relative complexities.

Section 3.3.1 details complexities that may arise in the Refine transformation due to interactions between specific members (versions, variants) of component and operator families. Constraints between specific objects make the Refine transformations more complex. These constraints are classified into three categories, based on what kinds of selections they constrain and which processes they affect.

The Compose transformation is discussed in Section 3.3.2. It details what complexities are introduced into the process during the refinement of a generic component into a concrete component (within a component model) and is able to alter the shape of the (component) model. This implies that information about concrete objects and their structural interconnections must be maintained. In order to manage the complexity of this new information, the concept of dynamically constructed relations is introduced.

Section 3.3.3 discusses the Merge transformation. It introduces input-sensitive operators that, when found in a configuration, make it impossible to know a-priori all of the specific derived objects that will be constructed. The Merge transformation is complicated by the fact that certain computations will have to be postponed until the configuration is partially manufactured.

3.3.1 The Refine Transformation

In the previous chapter, Refine transformations were used to refine generic models into specific models[1]. During Refine transformations, the structure of a generic model was fixed and refinements from generic objects to specific objects entailed selecting the appropriate member from within an object family, i.e., the one with the right attributes. While the Refine transformation was taking place, there were no constraints involved other than those supplied in the original request. In this section the Refine transformation still uses a fixed structure but new constraints can be introduced as specific objects are refined. These constraints complicate Refine transformations because the constraint set used to drive the transformation can change while refinements are still taking place. The effect is that it may no longer be possible to make choices independently.

As a software system evolves and changes, certain incompatibilities between specific versions of components and operators can arise. A particular version of one component can be incompatible with specific versions of other components. The use of a specific version of an operator can constrain the choice of other operators. A particular version of a component may require the application of a specific version of an operator. In order to manage this kind of complexity, it is necessary to associate specific constraints with specific objects. These constraints are called object-specific constraints.

Object-specific constraints complicate the Refine transformation. In the refinement processes described in the last chapter it was assumed that the selection of a concrete object from a family was determined exclusively by the requirements or constraints specified in the request for a product[2]. Because object-specific constraints are not supported, it is possible to perform each individual refinement independent of all others. With the inclusion of object-specific constraints, new constraints may be introduced during the Refine transformation that may affect subsequent and/or previous refinement selections.

In the next three paragraphs, three kinds of object-specific constraints are introduced: component-component, operator-operator, and component-operator. In each of these paragraphs an example of such a constraint is given, along with the SCM process it affects. It is also shown how the constraint affects refinement selection.

Sometimes in the transformation from a generic component model to a specific one, the selection of a particular concrete component demands that another specific concrete component must also be selected. This kind of constraint is said to be a component-component constraint. Component-component constraints complicate the Refine transformation used in the component selection process that transforms a gCM into a sCM (gCM-sCM transformation). An example of a component-component constraint is depicted in Figure 12 by the dashed arrow labeled (1) flowing between the Opt-Spec and Seman-Spec component families. The Opt-Spec(Experimental) object is

[1]This process is similar to one carried out in DSEE[46], in which system models are converted to bound configuration threads.

[2]A Pascal-Vax-Compiler is constrained to select "Pascal" and "Vax" family members.

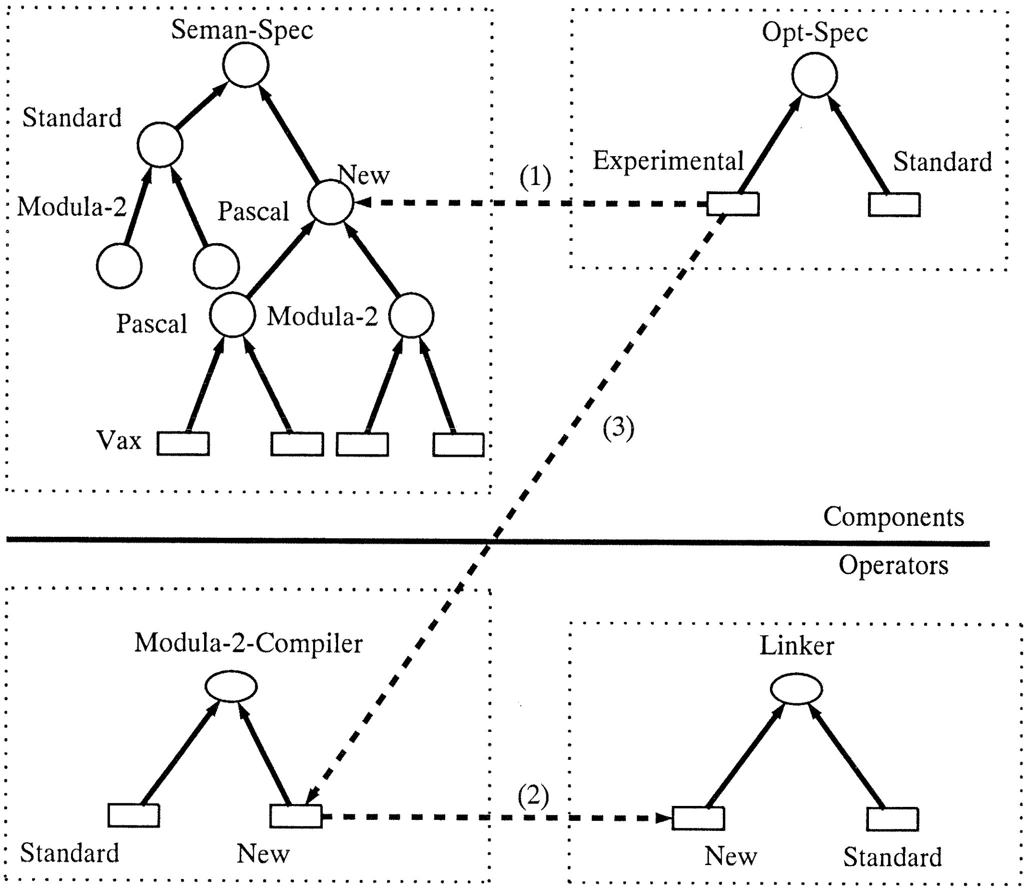

Figure 12: Constraint Example

an experimental version of a machine-independent optimization module. The use of Opt-Spec(Experimental) in a configuration demands that the "new" version of the Seman-Spec also be used (Seman-Spec(New)). The Seman-Spec(New) object introduces additional information into the IL that is needed to perform the new optimizations encoded in Opt-Spec(Experimental). If both the generic components Opt-Spec and Seman-Spec are included in a gCM and the Refine transformation selects Opt-Spec(Experimental), then the component-component constraint associated with this specific object will constrain the Refine transformation to select a Seman-Spec(New) version from within the Seman-Spec family[3].

Sometimes in the transformation from a generic operator model to a specific one, the selection of a particular concrete operator demands that another specific concrete operator must also be selected. This kind of constraint is said to be an operator-operator constraint. Operator-operator constraints make the Refine transformation from a gOM to a sOM (GOM-SOM transformation) more complex. An example of an operator-operator constraint is depicted in Figure 12 by the dashed arrow labeled (2) flowing between the the Modula-2-Compiler and Linker operator families. The Modula-2-Compiler(New) object produces a modified binary image format that cannot be understood by older linkers. The Linker(New) operator understands both the old and new binary image formats and can therefore be used for the output produced by either of the Modula-2-Compilers. Unfortunately, the Linker(New) operator is not fully tested and its use is limited only to situations where it is necessary (it is not used by default). If a gOM includes both the Modula-2-Compiler and the Linker generic operators and the Refine transformation selects the Modula-2-Compiler(New) object, then the operator-operator constraint associated with this specific object further constrains the Refine transformation to select the Linker(New) specific object from within the Linker operator family.

Sometimes the selection of a specific component in a configuration demands that a specific concrete operator must be selected. This kind of constraint is said to be an component-operator constraint. Component-operator constraints affect more than one transformation. This is due to the fact that the constraint affects two different kinds of objects, components and operators. In each of the last two paragraphs, because the constraints were restricted to elements of the same kind, only a single transformation was affected. Because there was no interdependence between these two transformations, no ordering was imposed on when they could be performed. For example, it would be possible to perform the gCM-sCM and the gOM-sOM Refine transformations in parallel. Component-operator constraints make the entire selection process more difficult because they introduce potential interactions between the component and operator Refine transformations.

An example of a component-operator constraint is depicted in Figure 12 by the dashed arrow labeled (3) flowing between the Opt-Spec component family and the

[3]This constraint does not determine the specific selection of a Seman-Spec component. Rather it constrains the selection to a sub-family within the family. For example, Seman-Spec(New,Pascal).

Modula-2-Compiler operator family. The experimental version of the Opt-Spec object, Opt-Spec(Experimental), uses facilities that can only be interpreted by the new version of the Modula-2 compiler (Modula-2-Compiler(New)). Assume that in the process of constructing a configuration, the gCM for that configuration includes an Opt-Spec generic component and that the gOM for that configuration includes an Modula-2-Compiler generic operator. If the gCM-sCM transformation selects the specific Opt-Spec(Experimental) object, then the component-operator constraint associated with that object must force the gOM-sOM transformation to select the Modula-2-Compiler(New) member from the Modula-2-Compiler operator family. As long as only component-operator constraints exist and not operator-component constraints, the resulting complications are minimized. By forcing the gCM-sCM Refine transformation to be performed before the gOM-sOM Refine transformation, it is possible to collect all of the operator constraints imposed by object-specific component-operator constraints associated with specific selected objects. These constraints can then be used to perform the gOM-sOM Refine transformation.

In the paragraphs above, three different kinds of constraints were introduced and the effect each one had on the selection of another object was described. These constraints are depicted in Figure 12. The remaining paragraphs show how these constraints interact when an attempt is made to satisfy an actual request.

Before describing these interactions, it is important to note that three assumptions are made. The first assumption concerns the shape of the Seman-Spec component family. As can be seen in Figure 12, a new level is introduced just below the root. This new level produces two sub-trees similar to the Seman-Spec family depicted in Figure 11 but differ in that each has an additional attribute with different values ("standard" and "new"). The second assumption is that a request is made to the system for an Experimental-Pascal-Vax-Compiler. The third assumption is that when processing the request, the initial constraints found in the request ("experimental", "Pascal", "Vax") are used to create a constraint-set that is utilized and maintained throughout the request transaction.

To satisfy the request for an Experimental-Pascal-Vax-Compiler, and thus to select a member of the Opt-Spec family, the initial constraint for an experimental compiler forces the Opt-Spec(Experimental) object to be selected. The Opt-Spec(Experimental) object has two constraints associated with it: a component-component constraint and a component-operator constraint. These constraints are added to the constraint-set.

If the Seman-Spec generic object has not yet been refined, then because of the object-specific constraint and the constraints given in the initial request, the Seman-Spec(New, Pascal, Vax) object is selected. If however, the Refine transformation has already taken place, the Seman-Spec(Standard, Pascal, Vax) object would have been chosen (via default selection and the initial constraints). As a consequence of the new constraint, the choice for the Seman-Spec object is no longer valid and must be refined again.

When selecting a member of the Modula-2-Compiler family, because the component-operator constraint associated with the Opt-Spec(Experimental) object has been added to the constraint-set, the Modula-2-Compiler(New) object must be selected. In doing so, the operator-operator constraint is added to the constraint-set.

Finally, if the Linker generic object has not yet been refined, then due to the new constraint the Linker(New) object must be selected.

3.3.2 The Compose Transformation

The Compose transformation is used in the generic component selection process, described in Section 2.2.1.1, to compose a gCM. Compose is also used in Section 2.2.1.2 to compose the gOM. In both cases, a generic model represents a fixed structure describing the set of generic components, and their interconnections, needed for all versions and variants of a software system. Generic models were then refined to specific models by instantiating the generic components contained in the generic model. The refinement from a generic to a specific model did not affect the shape of the model. In this section, the complications needed to support versions and variants of software systems whose structure can not be defined generically, are introduced. Because the shape of the model is a function of the particular version or variant desired, it is necessary for the process that composes a model to interact with the process that refines generic components into concrete ones.

During the Refine transformation described in the last section, constraints were introduced when specific objects were selected. The effect of these object-specific constraints was to further constrain the selection of other specific objects. During the Refine transformation it was assumed that the structure of the model was fixed—independent of the selections made within an object family.

However, what if the selection of a specific object also changed the shape of the structure being refined (i.e., demands the inclusion of one or more other objects). Models whose shape are a function of specific object selections are said to be selection-specific models. Selection-specific models are needed because unfortunately, generic models do not always apply to all members of a product family. The differences in models come about either because different variants demand different models or versions induced by change perturb an existing model.

Think for a moment about a compiler model in which one of the abstract components represents the compiler's interface to the underlying operating system (OS). Because all operating systems are not created equal some of the services supplied by one OS may have to be embodied in the compiler itself to be compatible with another OS. Supplying such services in the compiler might demand the inclusion of another module. Therefore, the shape of the model for the two compiler variants will be different.

Another example demonstrates the possible effects that evolutionary change may have on a component model. Say that the Opt-Spec module becomes too large and that it needs to be split into two modules. The shape of the model for this and all subsequent versions of a compiler will be different from all previous versions. Operator models suffer from a similar problem. As with a module that is split in two, a new version of a tool may be decomposed into two tools that share an intermediate object.

In systems that support selection-specific models, the Compose transformation and the Refine transformation become interdependent. Take an extreme case in which the generic operator model initially consists of a single generic component, that is, the root component for some software product. When the generic component is refined into a specific component, information associated with the object indicates that this particular concrete object is interconnected with two other generic objects. During the Refine transformation the Compose transformation is called upon to expand the previous model in order to incorporate them into the new model. At this point the model now contains three objects and two arrows. One of the objects has been refined and the other two have not. The Refine transformation then visits one of the other generic components and the cycle repeats itself. This process of composing and refining terminates when there no more objects to be added to the model, and all of the generic objects in the model are refined.

Maintaining the consistency of a static, manually-constructed structural relation is prone to error as a system evolves. Given the complexity of maintaining static information, methods for dynamically extracting these relations have been explored[71,40].

The difference between static and dynamic structural dependencies can be demonstrated by describing "include" dependencies. For example, in the C language[44], include statements direct the compiler to include the contents of other files before compiling. Typically include statements refer to objects called header files. Header files typically contain common and/or global declarations. SCM tools like Make[28] only support static structural interconnections. Each time an include statement is added to a source file or a header file[4], the static structure must be updated so the system knows about the new interconnections. If the static structure is not updated, these new interconnections are not included in the request satisfaction process and errors can be introduced into the resultant products. On the other hand, if these interconnections can be dynamically inferred when the request is made, the interconnections would be extracted from the source thus guaranteeing consistency.

In order to support dynamic structural relations, the Compose transformation may have to perform some kind of operator application to extract this information. This perturbs the Compose transformation by making it dependent on the Build transformation.

3.3.3 The Merge Transformation

The Merge transformation takes the sCM and the sOM and transforms them into a complete configuration which includes all of the derived objects produced by the operators specified in the configuration. This is possible if all of the operators are input-insensitive. An operator is said to be input-insensitive if the number of derived objects it produces is fixed, regardless of the values of its input(s). On the other hand,

[4]Include statements can be nested.

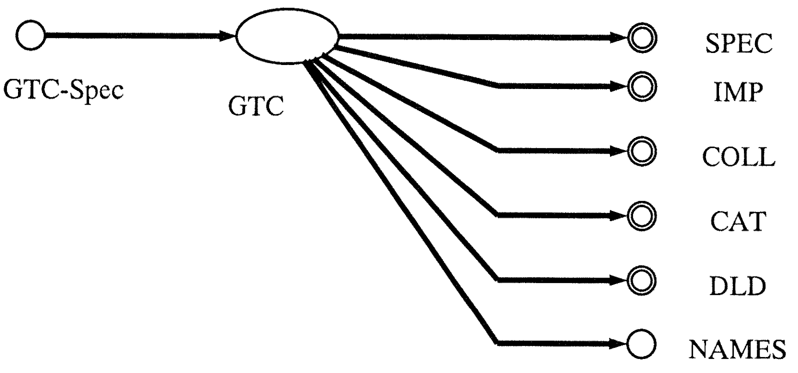

Figure 13: The GTC Operator

an operator is said to be input-sensitive if the number of objects produced is a function of the value of one or more of its inputs. The GTC operator found in Figure 13 is an example of an input-sensitive operator. It represents an abstraction for a data definition language compiler that was produced for testing certain concepts of the GT model.

The input to the GTC operator is a single object that contains a set of type specifications. When applied, the GTC operator produces five separate kinds of objects for each type defined in the input specification. All of the derived objects produced are programs or program fragments written in the E[26] language.

In Figure 13, the double-circles denote derived object sets or aggregates (as opposed to single derived objects). There are five of these circles, one for each of the following kinds of code fragments produced for each type defined in the GTC specification:

- A SPEC derived object that describes the type interface.
- An IMP derived object that describes the type implementation.
- A COLL derived object that creates a persistent container for members of that type.
- A CAT derived object that describes a procedure to create catalog information needed for the typed query interpreter.
- A DLD derived object that describes a function that creates dynamic loader entries so that types can be loaded while the system is running and their methods can be called by the query interpreter.

The sixth object, NAMES is a list of types defined in the GTC-Spec object. For example, if there were three type definitions in the input specification, then there would be sixteen derived objects produced.

Because the number of objects produced by input-sensitive operators is not statically known, the Merge transformation cannot determine all of the derived objects that will be produced. Moreover, as described in Section 3.4, subsequent derivations that emanate from these derived objects can not be detailed until after the actual elements of the aggregate of objects are known.

3.4 The Manufacturing Process

The manufacturing process described in Section 2.2.2 is broken down into two sub-processes, Schedule and Build. The schedule process uses a configuration as input, applies a Schedule transformation, and produces an operator schedule. The build process uses an operator schedule as input, applies the Build transformation which executes the operators in the schedule and produces the product. To support the basic SCM process these two transformations can be performed sequentially with no interactions. On the other hand, when complex predicates are allowed the Schedule and Build transformations must interact. Moreover when input-sensitive operators are allowed the entire manufacturing process must interact with the composition process.

In the basic SCM process, input-sensitive operators are not supported and the consistency predicate used is a simple time-stamp which permits the static construction of the operator schedule. All the information to produce a complete, static operator schedule is available. It is therefore possible to separate the Schedule transformation from the Build transformation. In addition, the manufacturing process as a whole is not intertwined with the composition of a complete configuration. As long as a complete configuration exists, the manufacture process has all the information it needs to execute in isolation.

In this section, the manufacturing process becomes more complex due to the introduction of complex predicates and input-sensitive operators. Complex consistency predicates require that intermediate computations be performed to determine operator consistency. To allow the use of such predicates, it is necessary to support dynamic operator scheduling. This implies the intertwining of the Schedule and Build transformations.

Input-sensitive operators complicate the Build transformation by causing interactions between the manufacture and composition processes. When a configuration contains an input-sensitive operator, the configuration produced in the composition process is incomplete and can only be filled in after the operator has executed. The act of filling the configuration is performed by the Merge transformation which, in turn, may rely on executing both the Compose and Refine transformations. Therefore, the introduction of input-sensitive operators also implies that the Build and merge transformations must somehow be intertwined.

The remainder of this section is broken down into two sub-sections. In Section 3.4.1 the Schedule transformation is described and the problems that complex predicates introduce are explained. In Section 3.4.2 the Build transformation is detailed and the problems that input-sensitive operators introduce are explained.

3.4.1 The Schedule Transformation

In the previous chapter, the process of constructing the operator schedule was completely separated from the execution of that schedule. For some predicates, this separation is impractical. Examples of complex consistency predicates are: value-equivalence, smart-recompilation[80], smarter-recompilation[73] and interface recompilation[62].

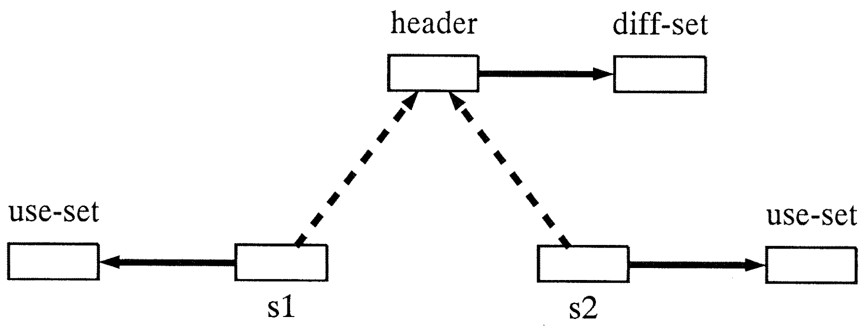

Figure 14: A Smart-Recompilation Example

Take for example smart-recompilation. Smart-recompilation is a complex predicate that subsumes the time-stamp predicate described in the last chapter. Smart-recompilation starts with the objects that the time-stamp predicate yields "inconsistent" and performs a finer-grained analysis on them. The information obtained by this analysis allows for a more accurate assessment of the objects actually affected[5]. Figure 14 depicts an example of the information needed to perform smart-recompilation. In this example, there are three objects: one header object and two source objects (s1 and s2). The header object contains variable declarations for FOO and BAR. The source objects s1 and s2 both contain include statements that make them dependent on the header object.

If the contents of the header object are changed and a product request that includes these three objects is made, then the time-stamp predicate determines that both source objects have to be re-compiled. As it turns out, the s1 object uses only the FOO declaration and the s2 object uses only the BAR declaration. Moreover, the change to the header object affects only the BAR declaration. The time-stamp predicate, because of its object-level granularity, does not have enough fine-grained information to make this observation and therefore compiles both source objects.

The smart-recompilation predicate was created to handle precisely this type of case. It does so by first computing additional information about objects as depicted in Figure 14[6]. Figure 15 shows the additional operators needed to produced this information.

The Compute-diff operator uses a header object as input and produces a diff-set. If the diff-set for a particular input object does not already exist, then the application

[5]The motivation behind smart-recompilation is that the cost of computing the additional information for an entire set of objects should cost less than the cost of one compile. If several compilations need not be performed, then the predicate is a big win.

[6]The description of the information and process for carrying out smart-recompilation is simplified here for explanatory purposes.

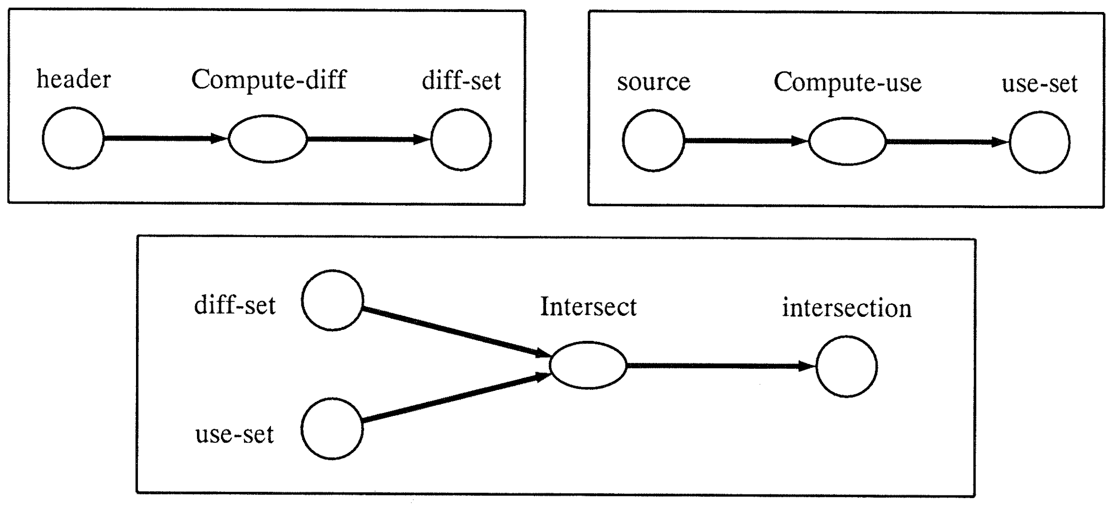

Figure 15: Aditional Operators f¡or Smart-Recompilation

of this operator results in the production of a diff-set object that contains all of the declarations defined in the input object. If the diff-object already exists, then the resultant diff-set object reflects only the declarations that have changed[7].

The Compute-use operator uses a source object as input and produces a use-set. The use-set contains all of the names of the declarations the source object imports[8].

The third operator, Intersect, uses a diff-set and a use-set as input and produces an intersection object that contains the set of names found in the intersection of the two sets. If the value of the intersection object is "empty", then the change in the object associated with this diff-set does not affect the source object associated with use-set. For example, if the header diff-set contained BAR and the use-set associated with the s1 object contained FOO, then their intersection would be "empty". An empty intersection implies that the change to the header file does not affect the s1 source object.

The first time a request is made for a product containing these objects, part of the operator schedule looks like the one shown in Figure 16. The Build transformation then performs all the operations in the schedule and the request terminates. Suppose that a change transaction was enacted on the header object that changed the declaration for the BAR variable. If the same request was made to the system, the schedule shown in Figure 17 would be statically produced. The following paragraph explains why.

[7]Actually more than one object must be produced: the first that includes the names of all declarations and the second that includes only the declarations that have been changed, added or deleted.

[8]Again, more than one object must be produced: the first that includes the names of all of the imports and the second that includes only the imports that have been changed, added or deleted.

INPUT	OPERATOR
header	Compute-diff
s1	Compute-use
s2	Compute-use
{s1-use,diff-set}	Intersect
{s2-use,diff-set}	Intersect
s1	Compile
s2	Compile

Figure 16: Operator Schedule using Smart-Recompilation

INPUT	OPERATOR
header	Compute-diff
{s1-use,diff-set}	Intersect
{s2-use,diff-set}	Intersect
s1	Compile
s2	Compile

Figure 17: Operator Schedule Using Smart-Recompilation

The smart-recompilation predicate uses the time-stamp predicate to determine that the header object has changed and that the s1 and s2 objects are "inconsistent" because they depend on it. This leads to the inclusion of the two Compile operators in the schedule. In order to determine whether those compilations really need to be applied the predicate must compute the intersection of the diff-set and the use-set which results in the inclusion of the two Intersect operators. To perform the Intersect operators, the header diff-set must first be computed.

Once the schedule is complete, it is then executed. The s1 object is not affected by the change to the header object. Because of this, the result of executing the first Intersect operator in the schedule (the one whose input set contains s1-use) is the production of an intersection object whose value is "empty". At this point the Schedule transformation is called. Based on the resulting intersection object, the Schedule transformation decides to remove the Compile operator whose input was s1, and all other operators that were exclusively and transitively included as a consequence of the inclusion of this operator from the schedule.

3.4.2 The Build Transformation

In the previous section complex consistency predicates were introduced that added complexity to the Schedule transformation. In this section, complexities are introduced into the Build transformation in order to handle input-sensitive operators. Input-sensitive operators make the build process more complex because it becomes necessary to call the Merge transformation at run time to fill in parts of a configuration.

In the rest of this section, an example that demonstrates the complexities introduced into the build process due to input-sensitive operators is developed. First a collection of simple tools that are used to satisfy a request are described. In this collection, a new kind of operator is introduced, one that takes a set (or aggregate) as input and produces a set (or aggregate) as output. An example is given of how such an operator is used. Then a request for a product that uses an input-sensitive operator is given. The final paragraphs discuss the actual building of the requested product. First the generic operator model for the request is constructed. Then the statically constructed configuration is discussed, including any missing information. As operators are executed and information becomes available, the Merge transformation must be called to complete the configuration.

Figure 18 shows a collection of simple operators, one of which is input-sensitive. The next three paragraphs describe these operators.

The ddl operator is an input-sensitive operator. It represents a simplification of the GTC operator discussed previously in this chapter. It uses as input a single ddl-spec object that contains a set of type definitions and produces a set of derived objects (ddl-il), one for each type defined in the ddl-spec. Because the ddl operator is input-sensitive, the number of derived objects produced is a function of the contents of the ddl-spec.

The ddl-comp operator is a new kind of operator. Both its input and output are sets. This kind of operator is called a set operator. The output is produced by applying

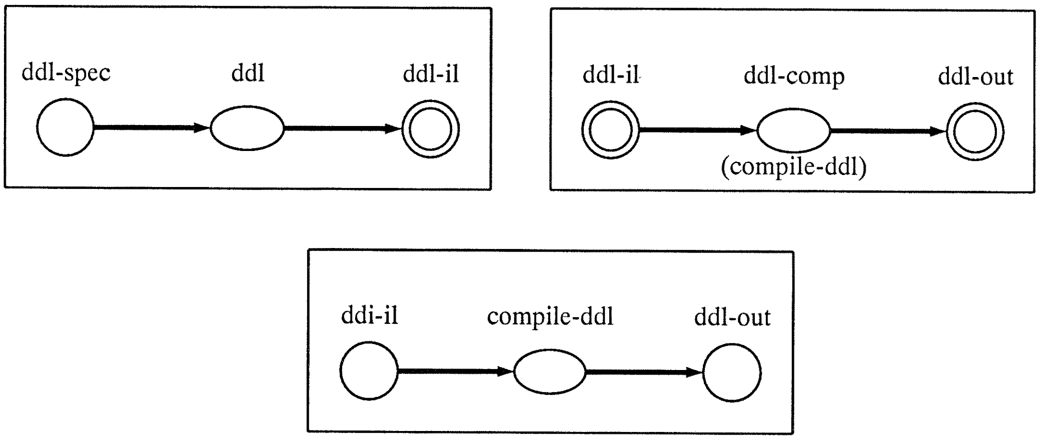

Figure 18: Operators for Build-Merge example

another operator to each of the members in the input set, producing an equivalent number of members in the output set[9]. The input to the ddl-comp operator is a set of ddl-il objects. The operator that is applied to each of these members is specified to be the compile-ddl operator described below. The result of the application of this operator is the creation of a set of ddl-out objects.

The compile-ddl operator uses a single ddl-il object as input and produces a single ddl-object.

It is possible to statically compose a complete configuration using set operators if the members of an input set can be statically determined. Suppose that one, a comp-set is defined to be a set of Modula-2 source objects. Two,that a set operator existed which took a set of Modula-2 objects and produced a set of binary objects. Three, that a linker operator existed that took a set of binaries as input and produced a single executable object. If a request is made to the system to produce an executable object from a specific comp-set object, then by following the basic process described in the last chapter, a complete configuration can be constructed. The details for filling in the application of the Modula-2-Compiler operator for each of the members of the comp-set is handled by the Merge transformation (see Section 2.2.1.3).

On the other hand, if the members of the input set to a set operator are not statically known—they are produced by an input-sensitive operator, then a complete configuration cannot be statically composed. The rest of this section details the satisfaction of a request that uses an input-sensitive operator.

Suppose that a ddl-spec object exists that is named "type-defs" that contains

[9]The "operator" could in fact be an arbitrary request for a product that is applied to each member in the set.

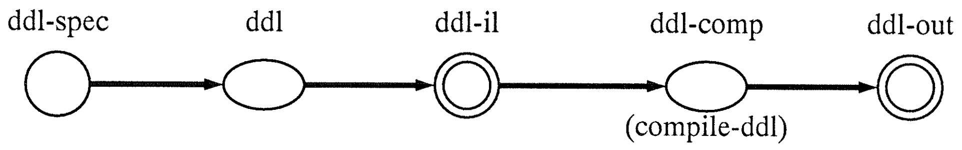

Figure 19: The sOM for the Build-Merge example

the description of the type definitions for two types: type-1 and type-2. Now suppose that a request is issued to the system to construct the ddl-out set of objects from the "type-defs" object. Producing the specific component model is simple because it contains only the "type-defs" object. Producing the specific operator is not much more difficult. Figure 19 depicts the specific operator model for producing a set of ddl-out objects from a ddl-spec. It states that a ddl-spec can be operated on by a ddl operator to produce a set of ddl-il objects and that this set can be operated on by the ddl-comp operator to produce a set of ddl-out objects. Producing the configuration is also a simple process. The result of the Merge transformation of the two specific models discussed above looks exactly like the specific operator model except that the name of the initial object "type-defs" is propagated where appropriate. Because the ddl operator is input-sensitive, the number of objects it produces is unknown. As a consequence, the configuration is incomplete and cannot be completed until after the ddl operator is executed.

In manufacturing this configuration, the Schedule transformation produces an operator schedule with two entries, one for the application of the ddl operator and one for the application of the ddl-comp operator. The Build transformation takes over and applies the operators in the schedule. The first operator applied is the ddl operator. The result of the application of this operator is the production of two ddl-il objects, one for type-1 and one for type-2. At this point the information needed to complete the configuration is available and the Merge transformation is called.

The result of the Merge transformation is the construction of a complete configuration as depicted in Figure 20. As can be seen in this figure, two compile-ddl operators are now merged into the initial configuration. Now that new operators exist in the request configuration, the Schedule transformation must also be called and results in the insertion of two compile-ddl operators directly below the ddl operator in the operator schedule. Control then returns to the Build transformation which carries out the application of the remaining operators in the schedule and makes the product.

3.5 Summary

In this chapter, some of the complexities that perturb the basic SCM process model described in Chapter 2 are introduced. These complexities come about because of versions and variants of objects, operator parameters, object-specific constraints,

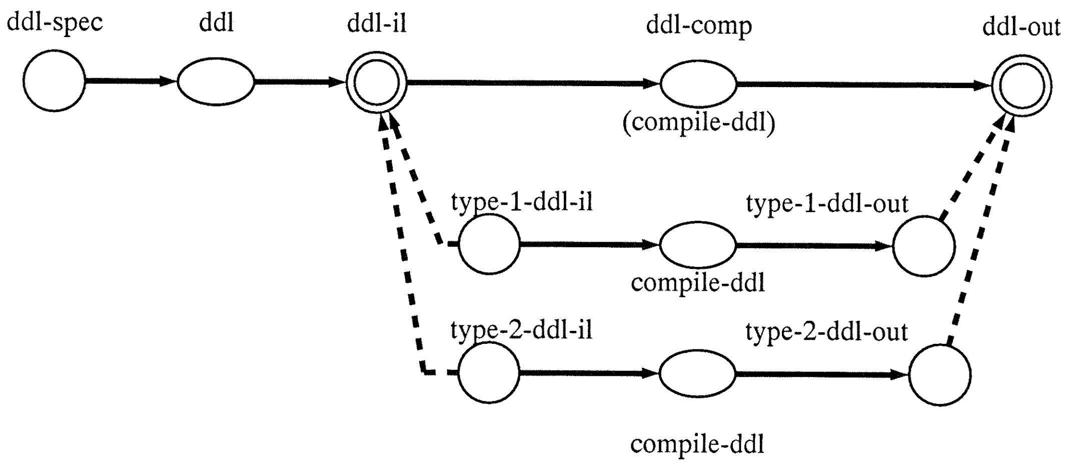

Figure 20: The Complete Configuration for the Build-Merge example

selection-specific models, complex consistency predicates and input-sensitive operators.

Each of these complexities, while expanding the range of problems that can be solved, also makes the process computationally more expensive. In software configuration management, as in most other computationally-complex processes, there are trade-offs that have to be made. The basic trade-off in software configuration management is between time and consistency.

CHAPTER 4

GRAPH TRANSFORM SCHEMAS

A good representation is explicit about the right things, constraint exposing, complete, concise, transparent, computationally efficient, detail suppressing, and computable — Patrick Winston[86].

Finding an appropriate knowledge representation is essential to producing a good solution to a problem. In the last two chapters, the software configuration management problem was explained in great detail. In this chapter, that information is distilled into concise representations which are then used in the next chapter to describe the SCM process in terms of graph transformations.

Components, operators, relations, constraints, operator graphs, and configurations are all important semantic entities for the SCM process. How they are specified, organized and manipulated determines the expressive power of a given configuration management system. In order to model a wide range of software configuration management systems it is necessary to provide a comprehensive and sufficiently powerful representation mechanism able to describe and construct such entities. In order to model the SCM process, one must be able to model and instantiate:

- **Components.** Components represent the objects produced and consumed by manufacturing steps.
- **Operators.** Operators encapsulate the behavior of manufacturing steps that operate on components.
- **Relations.** Relations are used to hold dependency information. For example, facts such as: "compiler.exe" **depends-on** "compiler.c", or that "FE-Spec" is **composed-of** "Lexer-Spec" and "Parser-Spec", are represented as tuples in relations.
- **Constraints.** Instances of constraints are associated with instances of components and operators in order to capture the concept of object-specific constraints introduced in Section 3.3.1.
- **Configurations.** Instances of configurations are produced by the composition process and consumed by the manufacturing process. Configurations, comprised of components, operators and relations, play a pivotal role in the GT model.
- **Operator Graphs.** Operator graph instances encapsulate system-wide knowledge about the kinds of components and operators defined and their interconnections. The operator graph is used extensively to infer the structure of generic operator models.

The remainder of this chapter is broken down into two sections. In Section 4.1, details of the Graph Transform Class specification language (GTC) and the basic classes and class constructors it provides, are introduced. Using the GTC specification language, Section 4.2 then defines the SCM-specific classes needed for the transformations, detailed in Chapter 5, that comprise the GT model.

4.1 The GT Specification Language

This section details the Graph Transform Class (GTC) specification language, and the initial class hierarchy it supports. These details are needed to understand the SCM classes defined in Section 4.2 and throughout Chapter 5.

GTC is a specification language developed to test the ideas and concepts found in the GT model. The GTC compiler uses a GTC specification as input and produces code written in the E[26] language that can be interpreted in an environment layered on top of the Exodus[10,11] database toolkit.

The interpreter is similar to a conventional relational query processor[1]. It is augmented with support for arbitrary, type consistent, method invocation (similar to SmallTalk[32]). A persistent collection (or container) is created to hold instances of each type (or class) defined, and a basic iterator is supplied for iterating through the collection. An Iterator[52] is a generalization and simplification of the iteration or loop mechanisms found in most programming languages. The basic iterator supplied, called Next, is a good example. Using the Next iterator, it is possible to produce an unordered stream of class instances. The iterator terminates when all of the elements in the class collection have been produced. In GTC one can also construct iterator methods for classes. For example, one could define a depth-first search iterator for a graph class and then use it as a method supplied to the query processor to yield the nodes of the graph in depth-first order.

The GTC language provides a few simple types, class specialization (inheritance) and aggregate class-generators to facilitate class construction.

As in most other object-oriented systems, these entities are encapsulated in a small, yet powerful, initial class hierarchy[4,70,50,32,59,43,2]. Figure 21 depicts the classes and class constructors initially provided by GTC. The remainder of this section is devoted to describing these entities.

The root object class in GTC is called Entity. Every Entity has at least three attributes: a name, a type and a unique object identifier (oid). The set of simple types provided by GTC consists of integer, string, boolean, and float. These types have the standard definitions found in most programming languages[44,41].

GTC also defines parametric aggregate class-generators for sets, lists and relations. These generators are specified in GTC as setof(BaseType), listof(BaseType) and relation(Domain, CoDomain), respectively. Aggregates are very useful for constructing complex classes. The setof generator supports the conventional methods typically associated with sets: Add , Remove, Find, First, Next, Member , Empty and Size. Next is a simple iterator that provides a convenient mechanism for yielding a stream of objects. The listof aggregate generator specializes the setof aggregate by ordering the elements in the set and supplies two additional methods Insert and Append.

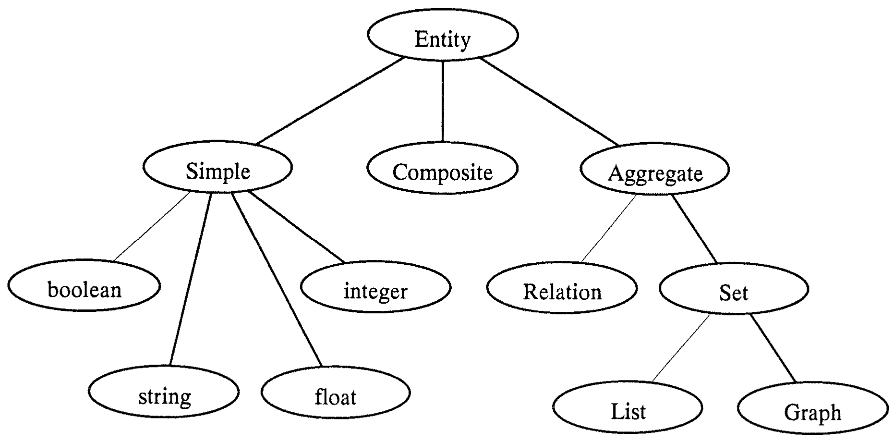

Figure 21: The class hierarchy provided by the GTC specification language.

Figure 22 depicts the specification for the relation class generator. The relation class supports the standard methods: Insert , Remove, Empty, Size, Union, Intersection and Difference, as well as iterators that yield transitive closure (Closure), depth-first search (DFS) and breadth first search (BFS). The Closure iterator method is used extensively in the GT model presented in Chapter 5.

4.2 Modeling SCM Semantic Entities

To understand the GT model described in Chapter 5, a small collection of SCM-specific classes are defined. The classes are defined using the GTC specification language described in Section 4.1.

The organization of the rest of this section is as follows. Section 4.2.1 defines the Composite class and its sub-classes Component and Operator. Section 4.2.2 introduces the Depends-on, Composed-of and Derives relations. Section 4.2.3 defines a simple Constraint class. The Configuration class, comprised of composite objects, relations and constraints is then defined in Section 4.2.4. Finally, Section 4.2.5 defines the Operator-Graph class, encapsulating the semantics of all the operators (and types) defined in a system.

4.2.1 SCM Composite Classes

The Composite class, specified in Figure 23, is the super-type for both the Component class defined in Section 4.2.1.1 and the Operator class defined in Section 4.2.1.2. The Composite class has three attributes: Consistency, Constraints and Depends. These

```
define  Relation[Domain,CoDomain]
   description:
     "Relation  aggregate generators for the GTC language"
   data:
   methods:
        void     Insert (Elem1 : Domain, Elem2 : CoDomain )
        void     Remove (Elem1 : Domain, Elem2 : CoDomain )
        boolean  Empty ()
        void     Union          ( r: Relation)
        void     Difference     ( r: Relation)
        void     Intersection   ( r: Relation)
        int      Size ()
        iterator Domain DFS()
        iterator Domain BFS(e : Domain)
        iterator Domain Closure(e : Domain )
end Relation;
```

Figure 22: The Relation Generator Class

```
define Composite
   description:  "Composite refines Class."
   specializes  Entity
   attributes:
     Consistency  : string init("undefined")
         =("undefined" | "inconsistent" | "consistent");
     Constraints : listof(Constraints);
     Depends     : setof(Composite)
  methods:
     boolean IsConsistent();
end Composite;
```

Figure 23: The Composite Class

attributes capture the concepts of object/operator consistency found in Section 2.2.2.3, object-specific constraints found in Section 3.3.1 and selection-specific models discussed in Section 3.3.2. The following paragraphs describe the details of the attributes and methods for this class as found in Figure 23.

Consistency. The Consistency attribute, initially set to undefined, can have a value of "undefined", "consistent" or "inconsistent". The three values correspond to values defined for object consistency in Section 2.2.2.3.

Constraints. The Constraints attribute serves as a container for object-specific constraints. It is defined as a list of the Constraint objects as described in Section 4.2.3. The Refine transformation discussed in Section 5.3.2, uses the contents of the Constraints attribute to guide the refinement of generic objects into concrete ones.

Depends. The Depends attribute is used for constructing selection-specific models. Defined as a set of (pointers to) Composite objects, its elements describe other objects that must be included in the model. For example, if version 4 of the system.c object (system.c-v4) demands that a version of system-aux.c also be included in the component model, then a pointer to the system-aux.c composite object is found in the Depends attribute of the system.c-v4 object. For SCM systems that support selection-specific models, the value of the Depends attribute is used by the Refine and Compose transformations (Section 5.3.2 and Section 5.3.2) to dynamically construct the relation defining the model's shape.

IsConsistent. The IsConsistent method is used by the Schedule transformation, discussed in Section 5.4.1, to aid in determining whether or not an operator needs to be (re)manufactured. It returns TRUE if the value of the Consistency Attribute is "consistent".

The Composite class and its sub-classes play an important role in the design of the Configuration class discussed in Section 4.2.4, and the OperatorGraph class defined in Section 4.2.5. In the following sub-sections the sub-classes of the Composite class are detailed. Figure 24 depicts the Composite class, and the hierarchy its sub-classes form.

4.2.1.1 Components

The Component class specified in Figure 25, is a sub-type of the Composite class. The Component class, serves as the root of all Component sub-classes and is useful when defining classes. It is the super-class for both the Primitive-Component and the Derived-Component classes. Primitive components are components that cannot be derived by an operator within the system, for instance, source files for a compiler. Derived components are produced by the application of operators. The Derived component class defines an additional two attributes discussed below. Derived components are further refined into simple and aggregate sub-types. Derived-Aggregates are used as inputs and outputs to Aggregate operators.

OperatorFor. The OperatorFor attribute points to the operator used to construct this derived component. It is used in the Build transformation, where the component will actually be constructed.

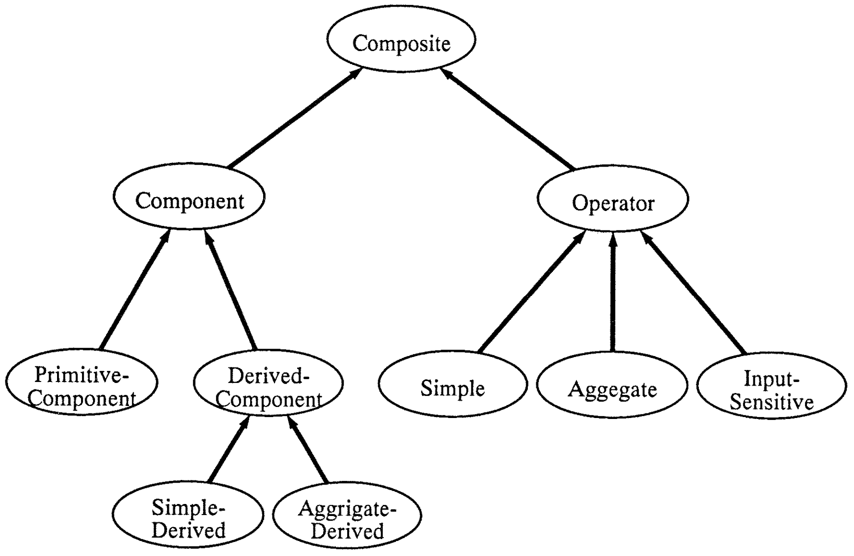

Figure 24: The Composite Class Hierarchy

Parms. The Parms attribute is used to define derived object families. Recall that in Section 3.2, derived component families were introduced because they provided a mechanism for describing, selecting and maintaining multiple versions of derived-objects. For example, with derived component families it is possible to simultaneously maintain "debugged", "standard" and "optimized" versions of a system. The Parms attribute has the same value as the Parms attribute of the operator pointed to by the OperatorFor attribute.

4.2.1.2 Operators

Operators are objects that encapsulate the input/output behavior of tools. The operator class has two sub-classes: simple and aggregate. The operator subclasses are discussed following the description below of the attributes and methods of the Operator class specified in Figure 26.

ToolFor. The ToolFor attribute describes the actual tool or tool script that is used when the operator is applied.

Inputs. The Inputs attribute defines the set of inputs or preconditions that must exist before this operator can be applied. Elements of the Inputs attribute are members of either the Primitive-Component or Derived-Component classes.

Output. The Output attribute represents the object produced by this operator. The Output attribute is restricted to designate a single Derived object. Since derived objects can be arbitrarily complex, this restriction does not affect the generality of the model.

```
%-----------------------------------------------------------
%       The Component Class
%-----------------------------------------------------------
define Component
    description: "Component class specializes Composite"
    refines Composite;
end Component;

    %-----------------------------------------------------
    % The Primitive and Derived  sub-classes
    %-----------------------------------------------------
   define Primitive-Component
      description: " Primitive specializes Component"
      refines Component;
   end Primitive;

   define Derived-Component
      description: "Derived specializes Component"
      refines Component;
      attributes:
         OperatorFor  : Operator;
         Parms        : listof(ToolParms);
      methods:
   end Derived;
```

Figure 25: The Component Class

```
%-----------------------------------------------------------
%       The Operator Class
%-----------------------------------------------------------
define Operator
  description: "Operator class definition"
  refines Composite;
  attributes:
     kind             : string;
     ToolFor          : ToolObject;
     Inputs           : setof(Component);
     Output           : Derived-Component;
     Parms            : listof(ToolParms);
     Delay            : boolean;
  methods:
    int apply(wait : boolean); /* must post to schedule */
end Operator;

    %-----------------------------------------------------
    % The simple, input-sensitive and aggregate subclasses
    %-----------------------------------------------------

    define Simple-op          specializes Operator   end;
    define Input-Sensitive-op specializes Operator   end;

    define Aggregate-op
       specializes Operator
       attributes:
          BaseType  : Composite;
          DeriveTo  : Operator;
       methods:
     end;
```

Figure 26: The Operator Class

Parms. The Parms attribute describes the set of parameters that the operator recognizes. Parameters are important for two reasons. One, they modify the behavior of the tool specified by the ToolFor attribute. Two, they define the attributes that comprise the derived object family. In fact, when the operator is defined, the Parms attribute is propagated to the Derived-Component class pointed to by the Output attribute.

Delay. The Delay attribute, initially set to FALSE, is used and manipulated in the Schedule and Build transformations described in Sections 5.4.1 and 5.4.2 respectively. When the Delay attribute is set to TRUE in an operator within a configuration, then the operator cannot be scheduled or built until additional information becomes available. Before configuration execution, the value of the Delay attribute is TRUE for any operator in a configuration that transitively depends on any input-sensitive operator.

Apply. The Apply method of the Operator class is responsible for constructing the appropriate environment for tool application, and then executing the tool described in the ToolFor attribute. Apply ensures that all the inputs are available (have values) and that the output object is produced.

The Operator class is further refined into three sub-classes: Input-Sensitive-op, Simple-op, and Aggregate-op.

Input-Sensitive-op operators are a sub-class of Operator in which the Output attribute is restricted to be a derived-aggregate component. Input-Sensitive-op operators were discussed in Section 3.3.3. Recall that the number and/or kind of objects produced by input-sensitive operators is a function of the object(s) used as input.

Simple-op operators are a simple sub-class of the Operator class. It defines no additional attributes or methods. For Simple-op, the type of the output attribute is restricted to the derived-simple class.

The Aggregate-op sub-class is a generalization of the set operator introduced in Section 3.3.3. Aggregate operators define two additional attributes described below. Aggregate operators provide a mechanism for applying an operator or operators to a collection of objects producing a collection of objects.

BaseType. The BaseType attribute specifies the type of elements found in the aggregate. Because of inheritance, all sub-types of this type can also be elements. For example, if the BaseType is Composite, then the elements could be of type Composite, or any of its sub-types Component, Primitive-Component, Derived-Component or Operator.

DeriveTo. The DeriveTo attribute specifies the product to be derived from each element in the input set.

4.2.2 SCM Relations

The relation aggregate class generator described in Section 4.1 is instantiated to reflect specific semantic properties of relations commonly used in the SCM process. The following three relations, depicted in Figure 4.2.2, are described below.

Depends-on is a general dependency relation between Composite objects. Composed-of is a specialization of Depends-on that describes structural interconnections. Derives is a specialization of Depends-on that describes derivational interconnections.

```
define Depends-on  relation(Composite,Composite) end;
   define Composed-of  specializes Depends-on end;
    define Derives     specializes Depends-on end;
```

Figure 27: Three SCM Relations

4.2.3 Object Specific Constraints

The Constraint class, depicted in Figure 28, are used to model object-specific constraints as detailed in Section 3.3.1. Constraint objects represent formulas in a first order predicate calculus. They are used to describe semantic properties of objects and the consistency conditions that must hold between them.

Constraints are specialized into three sub-classes one for each of the corresponding kinds of object-specific constraints discussed in Section 3.3.1: component-component, component-operator and operator-operator. This distinction is made to delineate the types of objects to which the constraints can be applied.

4.2.4 Configurations

As described in Chapter 2, the composition process transforms a user request into a configuration. The manufacturing process then uses that configuration to determine which operators, and in what order, need to be applied to satisfy the request. The configuration plays an important role throughout the entire SCM transaction. Instances of the Configuration class are manipulated extensively in all of the transformations described in Chapter 5.

A configuration is a special kind of graph that contains all of the Primitive-Components, the Derived-Components, the Operators (and their parameters), and the structural (Composed-of) and derivational (Derives) interconnections needed to construct a product. The root of a configuration indicates the initial component from which the product is to be derived. Conversely, the product of a configuration indicates the type

```
define Constraint
   description:"Represents a  GTM Constraint Formula"
   attributes:
        formula : string;     % textual representation of rule;
end;

    define component-component  specializes Constraint end;
    define component-operator   specializes Constraint end;
    define operator-operator    specializes Constraint end;
```

Figure 28: The Constraint Class

of object to produce from the initial component.

The following paragraphs describe the attributes and methods of the Configuration class depicted in Figure 29.

O-Nodes. The O-Nodes attribute represents the operators present in the configuration and is defined as a set of Operator-Nodes. Operator-Node objects have two attributes Refined and Node. The Refined attribute is manipulated by the Refine transformation found in Section 5.3.2. The Refined attribute is set to TRUE after the successful transformation of a generic object into a concrete one. The Node attribute is an instance of the Operator class defined in Section 4.2.1.2.

O-Edges. The O-Edges attribute represents the derivational relations that exist between operator and components within a configuration. It contains two kinds of edges. The first kind, from Components to Operators, defines Operator input behavior. The second, from Operators to Components, defines Operator output behavior. During composition, the O-Nodes and O-Edges together represent the stages of development of the operator model (gOM, sOM and points in between).

C-Nodes. The C-Nodes attribute represents the components present in a configuration. The attribute is defined to be a set of Component-Nodes. Like Operator-Nodes, Component-Nodes have two attributes Refined and Node. The Refined serves the the same purpose as in Component-Nodes. The Node attribute of a Component-Node is an instance of the Component class defined in Section 4.2.1.1.

C-Edges. The C-Edges attribute represents the structural relations that exist between components. During the various stages of the composition process, the C-Nodes and C-Edges together represent the various stages of development of the component model (gCM, sCM and points in between).

Root. The Root attribute specifies a Primitive component. It defines the root of the C-Edges Relation, and thus the component model. In the Vax-Pascal-Compiler example of Chapter 2, the Compiler-Spec component was considered the root component.

Product. In terms of an SCM request, a product is represented by a derived-component. But, because each operator has only one output object, the operator that produces the product is sufficient to uniquely determine the actual product. The operator is used instead of the actual derived component to facilitate the computations performed in the transformations discussed in Chapter 5.

InsertEdge. The InsertEdge method is used to add edges to models in the Compose transformation, detailed in Section 5.3.1. The parameters to InsertEdge are a Model string and a relation edge. The value of the model string is restricted to either "component" or "operator". When the value is "component", InsertEdge inserts nodes into C-Nodes if they do not already exist, and inserts the edge into C-Edges. When the value is "operator", InsertEdge inserts nodes into O-Nodes if they do not already exist, and inserts the edge into O-Edges.

BFS. The BFS iterator method is used by the Refine and Merge transformations to perform a breadth-first, left-to-right walk of the nodes in a model. The parameters to BFS are a Model and a Domain. The Model parameter is of type string and the Domain

```
%-----------------------------------------------------
%       The Configuration Class
%-----------------------------------------------------
define Configuration
   description: " Configuration Class Definition"
   attributes:
     O-Nodes     : setof(Operator-Node);
     C-Nodes     : setof(Component-Node);
     O-Edges     : Derives(ConfigNode,ConfigNode);
     C-Edges     : Composed-of(Component-Node,Component-Node);
     Root        : Component;
     Product     : Operator;
   methods:
     void InsertEdge(model    : string,
                     TheEdge  : Edge);
    iterator ConfigNode BFS(Relation : string,
                           Domain   : Composite)
    boolean  InsertOperator(op:Operator);
end;

   define Edge
      attributes:
        Domain   : Composite;
        CoDomain : Composite;
   end;

   define ConfigNode
      attributes:
         Refined : boolean;
         Node    : Composite
      methods:
   end;
   define Operator-Node   specializes ConfigNode end;
   define Component-Node   specializes ConfigNode end;
```

Figure 29: The Configuration Class

parameter is of type Composite. When the Model argument is "component", the Domain object and the elements yielded by the iterator are members of the Component sub-class. Conversely, when the model argument is "operator", the Domain object and the elements yielded by the iterator are members of the Operator sub-class.

InsertOperator. The InsertOperator method inserts an operator into a configuration. Besides using the operator argument to construct an Operator-Node and inserting it into the O-Nodes attribute, it must also add elements to the O-Edges attribute, according to the input/output semantics specified by the operator.

4.2.5 The Operator Graph

Operator graphs play a key role in the construction of generic operator models. They encapsulate all of the knowledge about the types and operators defined in a given SCM system, including the dataflow interconnections between them as defined by the Inputs and Output attributes of the operators contained therein. This information is coalesced into one object so that derivation paths between types can be automatically inferred. For any complicated manufacturing process this is essential. The OperatorGraph is designed in such a way that its FindPath iterator method carries out the complex inferencing needed to construct gOMs. The FindPath iterator method yields to its clients a stream of Derives edges, and looks exactly like the Closure iterator method supplied by the relation class generator discussed in Section 4.1. The reason for this is discussed in Section 5.3.4.

Figure 30 shows the OperatorGraph (OG) class definition. Patterned after Odin's Derivation Graph[12], it contains information about components and operators, and the relations between them.[1] The OG represents a forest of directed, possibly cyclic, graphs. The nodes in the forest represent all of the types currently defined in a system. The edges represent relations between those types. The three relations that exist over types within an OG are derives, super-type and equivalence. Figure 31 is a pictorial representation of a simple OG. The solid arrows denote derives relations, the dashed arrows denote super-type relations and the dotted arrow denotes an equivalence relation. The next three paragraphs describe the semantics of the arrows.

Derives. The Derives relation captures the concept of operator application. Derives edges come in two flavors: input and output. The input derivation relation embodies the relation between an operator and its inputs. For every input object defined in an operator specification, there exists a directed edge in the input derivation relation from that object to the operator. The edges from f-src to f-compiler and from c-src to c-compiler are examples of this relation. The output derivation relation embodies the relation between an operator and its output. For every operator there exists one output type. Therefore, there is a directed edge in the output derivation relation from the operator to the derived type. The edges from f-compiler to f-bin and from c-compiler to c-bin are examples of this relation. Together, the two derives relations capture the

[1]The distinction between the OG presented here and Odin's Derivation Graph is made clear in Section 6.4.

```
%-----------------------------------------------------
%        The OperatorGraph Class
%-----------------------------------------------------
define OperatorGraph
   description: "The Operator Graph"
   attributes:
     C-Nodes       : Setof(Primitive);
     D-Nodes       : Setof(Derived);
     O-Nodes       : Setof(Operator);
     I-derives     : Derives(Component,Operator)
     O-derives     : Derives(Operator,Derived-Component)
     super-type    : relation(Derived-Component,
                              Derived-Component)
     equivalence   : relation(Composite,Composite)
   methods:
      boolean        Consistent()
      int            InsertOperator(o : Operator)
      int            DeleteOperator(o : Operator)
      boolean        Insert(e1 : Composite, e2 : Composite);
      Operator       FindOperator(e : Edge)
      iterator Edge  FindPath(Src : Component , Dest: Operator)

end OperatorGraph;
```

Figure 30: The OperatorGraph Class

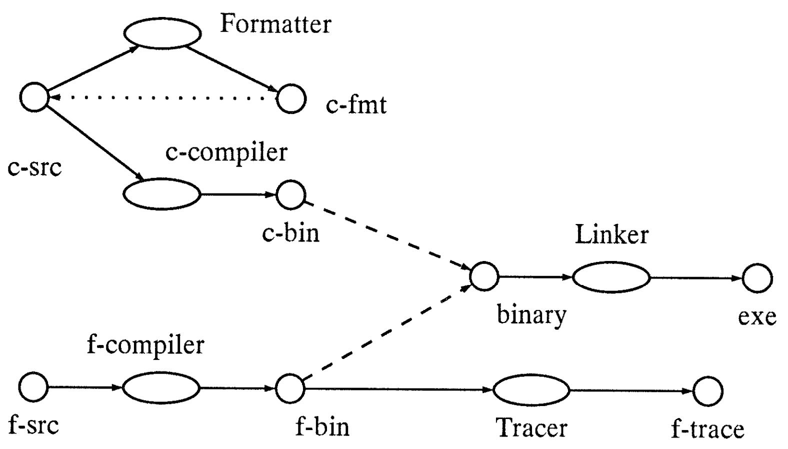

Figure 31: A simple Operator Graph

input/output semantics of operators. They also capture derivational dependencies needed for scheduling and building. For example, it is not possible to schedule or execute the Tracer operator until after the f-compiler operator completes.

Super-type. The super-type relation in an operator graph captures the concept of "type inheritance" for Derived-Component types. This is useful for collecting the common behavior of a set of types and reduces the number of derives edges that need to be specified. The directed edges from c-bin to binary and from f-bin to binary are examples of this relation. The semantics that these edges convey is that anything derivable from a binary object is also derivable from either an f-bin or a c-bin object as well. However, this relation does not say that a c-bin and an f-bin are semantically equivalent. For example, it is not possible to build an f-trace object from a c-bin object.

Equivalence. The equivalence relation is used to define functional equality. The edge from c-fmt to c-src is an example of this relation. Its semantics state that for any operator that uses a c-src object for input, a c-fmt object can be substituted.

In the remaining paragraphs, the attributes and methods of the OperatorGraph Class are described.

The C-Nodes, O-Nodes and D-Nodes attributes comprise the nodes of the operator graph. They respectively represent the sets of Primitive-Components, Operators and Derived-Components defined in the system.

I-Derives, O-Derives, super-type and equivalence attributes are all defined as relations and represent the three relations discussed above.

The InsertOperator and DeleteOperator methods are used to insert operators into, and delete operators from, the operator graph. Besides inserting an element into the O-Nodes attribute, the InsertOperator method must also use the information in the operator to update the two derives relations, and possibly add nodes to the C-Nodes and

D-Nodes attributes. Inversely, DeleteOperator removes an operator from O-Nodes and removes its associated edges in the derives relations.

FindOperator. The FindOperator method is used to find an operator in OG. It does so by looking in the O-Derives relation for the edge argument. It returns the operator associated with this edge. FindOperator is used in the Merge transformation described in Section 5.3.5 to perform edge labeling similar to that found in Make[28].

FindPath. The Findpath method is an iterator; it yields a stream of edges. The parameters to Findpath are Src and Dest. The Src parameter is of type Component and can be either of the Primitive-Component or Derived-Component sub-classes. The Dest parameter is an Operator. FindPath, using all of the relations within the OG, looks for a legal path from Src to Dest. The stream it returns contains those edges found along that path that are from either the I-Derives or the O-Derives relations.

Consistent. The Consistent method checks to see if the operator graph is in a consistent state. The OG must be in a consistent state for FindPath to guarantee that all legal paths can be found. A consistent operator graph is a necessary, but not sufficient, condition to guarantee the consistency of the transformations developed in Chapter 5. OG consistency is defined as follows. Let the OG be represented by the 2-tuple $\langle$ V, E $\rangle$ where:

- V is the set of Composite types defined in the system.
- E is the set directed labeled edges of the Derives relation.

Let Ω represent the set of operators defined in the system. Let $\omega = \langle$ N, I, O, T $\rangle$ represent an abstraction of an instance of the Operator class defined in Section 4.2.1.2 where:

- N is the name of the operator.
- I is the set of input objects, in which the elements are all of type Component.
- O is the output object, and is of type Derived-Component.
- T is the tool encapsulated by the operator.

Then, an OG is said to be consistent iff:

(1) $\forall$ p $\in$ *Primitive* [p $\in$ V]. Every Primitive component is an element of the vertex set.
(2) $\forall$ d $\in$ $Derived-Component$ [d $\in$ V]. Every derived component is an element of the vertex set.
(3) $\forall\, \omega \in \Omega$ [$\omega \in$ V]. Every operator is an element of the vertex set.
(4) $\forall \omega \in \Omega, \forall \iota \in \omega$.I [$\iota \in$ V, ω.O $\in$ V]. Every input type and output type specified in every operator belongs to the vertex set of the graph.
(5) $\forall \omega \in \Omega, \forall \iota \in \omega$.I [$\iota \rightarrow \omega \in$ E, $\omega \rightarrow \omega.O \in$ E]. For every operator there must exist a Derives edge from every input object to the operator, and from every operator to the output object.
(6) $\forall \omega \in \Omega$ [ω.O $\neq \perp$]. There does not exist an operator in which the output type is undefined.
(7) $\forall \omega \in \Omega, \forall \iota \in \omega$.I [$\iota \neq \perp$]. Every input of every operator is defined.
(8) $\exists \omega \in \Omega, \forall \iota \in \omega$.I [$\iota \in$ *Primitive*]. There exists at least one operator signature

whose input set contains only Primitive components.

(9) $\forall d \in Derived - Component[reachable(d)]$. For every Derived-Component type there exists a path in the OG to construct it.

4.3 Summary

This chapter distilled the information detailed in Chapters 2 and 3 into concise knowledge representations that are used extensively in Chapter 5. Representations for components, operators, relations, constraints, operator graphs, and configurations were specified and explained.

CHAPTER 5

THE GRAPH TRANSFORM MODEL

In this Chapter, the Graph Transform (GT) model for the software configuration management process is presented as a sequence of transformations on directed acyclic graphs. Figure 32 depicts the SCM process covered by this model. It is a terse graphical representation of the basic process described in Chapter 2, with the complexities introduced in Chapter 3. The dotted boxes denote processes. The parallelograms denote graph transformations. The circles represent software objects. The solid arrows denote the data and control flow for the basic model described in Chapter 2. The dashed arrows denote exceptions to the basic control flow that arise because of the exceptions discussed in Chapter 3. The labels on the dashed arrows (given in parentheses) describe the reason for each exception.

This chapter details each of the processes, transformations and exceptions of Figure 32 in terms of graph transformations and is organized as follows. Section 5.1 discusses the environment in which the GT model executes. Section 5.2 explains how a request is transformed into a graph representation. Section 5.3 describes how this graph is then used as input to the composition process, which applies the Compose, Refine and Merge graph transformations and produces a configuration. Then, in Section 5.4, the manufacturing process uses the configuration graph as input, applies the Schedule and Build graph transformations, and produces the product. The manufacturing process performs the "minimal" number of computations needed to produce a consistent product. Finally, a summary of the GT model is given in Section 5.5.

5.1 The Environment

Object-oriented classes were introduced in Chapter 4 to represent SCM knowledge. In this chapter, transformations that use these representations are developed. Before describing the details of the transformations, the details of the language in which they are written and the environment in which they operate are discussed.

The pseudo-code (language) used to depict the transformations in this chapter can be thought of as a hybrid of functional, logic and object-oriented concepts, as found in a language such as Goguen's FOOPlog[31]. The best way to think of this language is to start with a conventional object-oriented language and then unify it with a relational or logic programming language by adding logic variables and backtracking on failure.

The object-oriented language used in these transformations is based on E[26], a derivative of C++[75]. E was chosen because it supports object persistence, persistent collections, higher order class generators (parametric classes) and iterators, in addition to conventional object-oriented concepts. The "FAIL" keyword, similar to the fail goal in Prolog[15], is added to the language and causes the environment to backtrack. Every

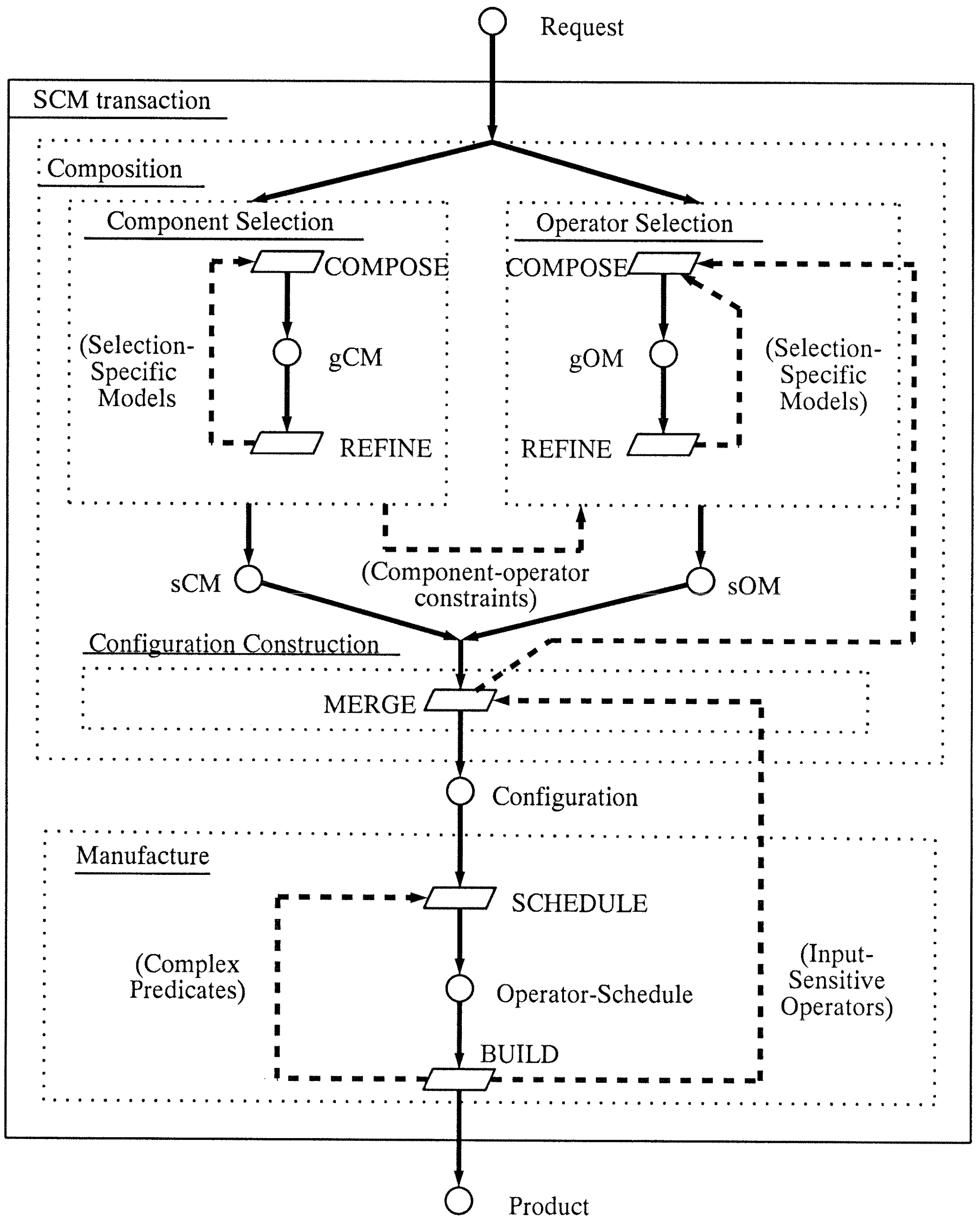

Figure 32: The SCM process

```
define RequestGraph
   description: "Working Graph for GT  request satisfaction"
   attributes:
     Config      : Configuration;
     CS          : ConstraintSet;
     OpSched     : OperatorSchedule;
   methods:
     int Union(other: RequestGraph)
end;
```

Figure 33: The RequestGraph Class Specification

time a decision or assumption is made using constraints or predicates (all written in a logic subset of the language similar to Prolog), information about the decision is recorded and used for controlling backtracking.

The environment in which these transformations operate is interpretive. As a consequence, the type and sub-type of each object is determined dynamically. One should think of the SCM process as a program over these transformations. Existing SCM systems will be described in this way in Chapter 6.

Because the SCM process is represented as a program executed by an interpreter, it is possible to change the program between requests, thus affecting the SCM policy being adhered to. At the end of this chapter, it should be clear that by manipulating the parameters and the sequencing of the transformations presented, different SCM policies can be enforced.

5.2 The Request Graph

The SCM transaction starts with a request for a software product. A typical request contains a root component, the desired product, and a (possibly empty) set of constraints. In the Pascal-Vax-Compiler example of Chapter 2, the root component was "Compiler-Spec", the product was "executable" and the set of constraints included "Language=Pascal" and "Machine=Vax". Requests are made to the SCM system in a textual form. The request is then converted into an internal representation called a Request Graph (RG). The RG is the working graph of the GT model, and is manipulated by the various transformations described below.

The RG is composed of a configuration, a constraint set and an operator schedule. Figure 33 depicts the specification for the RequestGraph class. The configuration is an instance of the Configuration class detailed in Section 4.2.4. The constraint set is an instance of the ConstraintSet class described below. Discussion of details of the OperatorSchedule class are deferred until Section 5.4

The ConstraintSet class specified in Figure 34 encapsulates the concept of constraint satisfaction[18,9]. Its attributes "Constraints", "Logic" and "Check" represent a

```
define ConstraintSet
   description: "ADT for a Constraint Satisfaction System"
   attributes:
     Constraints : setof(Constraint);
     Logic       : string = {"monotonic" | "non-monotonic"} ;
     Check       : boolean;
   methods:
     boolean   Apply(Configuration-Node : ConfigNode);
     void      Add(elements: Constraints);
     boolean   Satisfiable();
end ConstraintSet;
```

Figure 34: The Constraint Class Specification

set of constraints, a flag to determine which kind of logic to use, and a flag to determine whether or not to check the satisfiability of the constraint set, respectively. The elements of the Constraints attribute are members of the constraint class defined in Section 4.2.3. The Logic attribute, whose value can either be "monotonic" or "non-monotonic", describes the type of logic to be applied after the addition of a new constraint to the constraint set. The value of the Check attribute determines which action the Satisfiable method performs. When the Check attribute is FALSE, the Satisfiable method does nothing. When the Check attribute is TRUE, the Satisfiable method checks the internal consistency of the constraint set to determine if the set is satisfiable.

When the value of the Logic attribute is "monotonic", monotonic logic is used. This implies that the addition of a new constraint to the constraint set does not affect prior deductions. It also implies that the new set of deductions that can be made is a superset of those that could be made prior to the addition[60].

If, on the other hand, new constraints can invalidate previous assumptions, then default reasoning must be used. In order to use default reasoning, a non-monotonic logic is needed. Default reasoning is defined by Moore as: "the drawing of plausible inferences from less-than-conclusive evidence in the absence of evidence to the contrary"[61]. Using default reasoning, it is possible that as object-specific constraints are encountered they may introduce information that contradicts previous selections. As a consequence, it is necessary to backtrack to the point in the computation where the false assumption was made, select a new assumption, and restart the computation.

When the request is converted into an RG, the Root and Product attributes of the configuration are initialized, and the set of initial constraints is added to the constraint set. The following code fragment depicts how the Pascal-Vax-Compiler request would be converted into an internal representation:

```
   RequestGraph.New(RG),
   RG.Config.SetProduct(executable.OperatorFor),
RG.Config.SetRoot(Compiler-Spec),
RG.CS.Constraints.Add("Language=Pascal"),
RG.CS.Constraints.Add("Machine=Vax"),
```

5.3 Composition

In this section, the composition process depicted in Figure 32 is presented as a series of transformations that start with a RG and terminate when the configuration in the RG is composed. The basic process is similar to the one outlined in Chapter 2. It is composed of three sub-processes: component selection, operator selection and configuration construction. The transformations used in the composition process are: Compose, Refine and Merge.

The organization of this section is similar to the organization of the composition process depicted in Figure 32. In Section 5.3.1, the Compose transformation is discussed. It changes the shape of the RG by adding nodes and edges. In Section 5.3.2, the Refine transformation is discussed. It refines generic objects in a graph by instantiating attribute values. Section 5.3.3 explains how components are selected to form generic and specific system models by applications of the Compose and Refine transformations. Section 5.3.4 explains how operators are selected to form generic and specific operator models in a fashion similar to component selection. Finally, Section 5.3.5 discusses the Merge transformation that melds the specific component and operator models into a configuration. Possible interactions of the Merge transformation with the Compose and Refine transformations is also explained.

5.3.1 The Compose Transformation

The Compose transformation changes the shape of the RG by adding the appropriate nodes and edges to the configuration[1]. Figure 35 depicts the pseudo-code for this transformation. It is used in both the component and operator selection processes. In this section, the transformation is described in general terms. In Section 5.3.3, when more context is available, the Compose transformation will be re-visited.

The formal parameters to the Compose transformation are an RG, a Model string, a Relation, and an Iterator. The RG is an instance of the RequestGraph class described above. The Model parameter determines which model is being composed. When the value is "component" a component model (gCM) is constructed. When the value is "operator" an operator model (gOM) is constructed. The Relation parameter , also called the relation of interest, is an instance of the Depends-on relation, or one of its subclasses. The relation of interest is the source of information used for adding nodes and edges to the configuration contained in the RG. Information is extracted from the relation by using the **Iterator** parameter. The Iterator is a method used to produce a stream of edges. For component composition, the iterator is typically the Closure method of the relation aggregate as defined in Section 4.1. For operator composition, it is typically the FindPath method of the OperatorGraph class defined in Section 4.2.5.

[1]In the thesis proposal this transformation was called Extension.

```
Compose (
    RG : RequestGraph, /* the request graph */
    Model :  string, /* component or operator */
    Relation :  Relation, /* the relation to extend by */
    Iterator :  iterator) /* used to extract members*/
{
    RetCode = 1;
    iterate (edge = Relation.Iterator())
    {
        RG.Config.InsertEdge(Model,edge);
        RetCode = 0; /* RG is modified */
    }
    return RetCode;
}
```

Figure 35: The Compose Graph Transformation

To simplify the transformation, the arguments to the iterators are bound at the locations where the Compose transformation is called. This is done because different iterators may use different numbers and kinds of arguments.

The Compose transformation is used to compose models. It is used in both the component and operator selection processes detailed in Sections 5.3.3 and 5.3.4. The computations involved in this transformation are deceptively simple. It performs an iteration loop in which the Iterator is called to produce a stream of edges from the relation. As the edges are extracted from the relation of interest, they are inserted into the configuration. But, as detailed in Sections 5.3.3 and 5.3.4, depending on the actual relation and iterator parameters used, and the sequencing of this transformation with the others defined in this chapter, the Compose transformation becomes formidable The Compose transformation either returns 0 or 1. If the transformation actually augments the RG then Compose returns 0. The return code is used to terminate recursion in the dynamic selection process described in Section 5.3.3,

The Compose transformation is used to compose generic models (the gCM and the gOM) in configurations using static relations. It is also used to aid in model composition where the model is selection-specific. In this case, the Compose transformation will interact with the Refine transformation as discussed in the next section, causing the dynamic construction of the relation of interest.

5.3.2 The Refine Transformation

The Refine transformation is used to refine generic objects within models. It does so by using the constraint set to instantiate undefined attribute values[2]. Like the Compose transformation, Refine is used in both the component and operator selection processes. Refine relies heavily on the ConstraintSet object found in the RG for determining attribute values. Figure 36 depicts the Refine transformation.

The formal parameters to the Refine transformation are an RG, a Model string, a Relation and a Node. Refine modifies the values of the attributes of the nodes in the RG. The Relation parameter specifies the relation of interest. The possible values of Model parameter, as in the Compose transformation, are either "component" or "operator". The Node parameter specifies the location from which to start the refinement process. When the value of the Model parameter is "component" the type of the node will be the component sub-type of the Composite class. When the value of the Model parameter is "operator", the type of the node will be the operator sub-type.

Refine iterates over all of the nodes in a particular model and performs attribute instantiation. It checks to see if this node has already been refined. If not, it applies the constraints found in RG.CS.Constraints to the node. If the application of the constraint set is not successful, then the transformation fails and dependency-directed backtracking takes over. If constraint application succeeds, i.e., all of the attributes are assigned values and a unique concrete object has been found, then the Constraints and the Depends attributes of the object are inspected. Recall that in Section 4.2.1, the definition of

[2]In the thesis proposal this transformation was called Attribution.

```
Refine(RG : RequestGraph,
    Model :  string ,
    Relation :  Relation,
    Node :  Composite)
    {
        obj = Node;
        iterate ( config-node = RG.Config.NextNode(Model,obj))
        {
            obj = config-node.node;
            if ( (config-node.GetRefined() = FALSE) and
                RG.CS.Constraints.Apply(obj) )
            {
                config-node.SetRefined(TRUE);
                if not(obj.Constraints.Empty())
                { /* CASE 1 */
                    RG.CS.Constraints.Add(obj.Constraints);
                    if not(RG.CS.Satisfiable()) FAIL
                    if not(RG.CheckConsistency()) FAIL
                }
                if not(obj.Depends.Empty())
                { /* CASE 2 */
                    iterate ( dep = obj.Depends.Next())
                    {
                        Relation.Insert(obj,dep)
                    }
                }
            }
            else FAIL
        }
    }
```

Figure 36: The Refine Graph Transformation

the Composite class included a Constraints and a Depends attribute. The Constraints attribute contains object-specific constraints, and the Depends attribute contains the objects used for selection-specific models.

If object-specific constraints exist, they are added to the RG constraint set. The constraint set is then checked to see if it is still satisfiable. The configuration is also checked to see if any of the previous refinements are invalidated by the new constraints. Whether or not these two actions perform any computations depends on the state of the Logic and Check attributes of the ConstraintSet class used to satisfy the request.

The value of the object's Depends attribute is also inspected. Recall that in Section 4.2.1, the Depends attribute specifies the set of (generic) objects needed for a selection-specific model. If the Depends attribute is not empty, then the Insert method of the relation of interest is called to insert tuples.

The Refine transformation is used to refine objects within models. Constraints, both initial and object-specific, are used to constrain the possible choice of attribute values. Refine is used in both the component and operator selection processes detailed below. For SCM systems in which generic models are supported, Refine will be used to transform all of the generic objects in the model into concrete ones. In SCM systems where selection-specific models are supported, the Refine transformation interacts with the Compose transformation, dynamically constructing edges of the relation of interest.

5.3.3 Component Selection

For the basic SCM process, component selection is comprised of determining the generic component model and then refining the model into a specific one. In GT terms, the request graph is augmented using the Compose transformation with nodes and edges from a static Composed-of relation (in which the domain and the co-domain are both components). The Refine transformation is then applied to the gCM , instantiating attribute values of the nodes contained therein. For complex composition processes, the Compose and Refine transformations must be interleaved so that their intermediate results can interact.

The remainder of this section is devoted to describing four cases, each increasing the computational complexity of the composition selection process. The increase in complexity is a function of the number and kind of exceptions present. The exceptions that can affect component selection are object-specific constraints and selection-specific models. The four cases are:

- Case 1: A generic model.
- Case 2: A generic model with object-specific constraints.
- Case 3: A selection-specific model.
- Case 4: A selection-specific model with object-specific constraints.

Case 1. Recall that the basic process of Chapter 2 does not support either object-specific constraints or selection-specific models. Because selection-specific models are not supported, it is possible to construct generic component models. The information needed to construct the gCM is found in a static instance of the Composed-of relation. For example, imagine an instance of the Composed-of relation called "Compiler-Model" that contains the edges of the compiler model depicted in Chapter 2, Figure 1. Because object-specific constraints are not present, there is no way to modify the initial set of constraints. This implies that there are no interdependencies between individual object refinements within a given model. As a consequence, it is possible for the basic composition process to perform all of the refinements in parallel.

Figure 37 shows a program fragment that depicts the basic component selection process consisting of the Basic_Component_Selection procedure and the select-basic sub-procedure. The Basic_Component_Selection procedure calls the select-basic sub-procedure with the appropriate arguments. The arguments are an RG constructed in Section 5.2, an instance of the Composed-of relation containing the generic model and the Composed-of.Closure() iterator method to extract the model bound with the Root of the configuration.

In the select-basic sub-procedure, Compose is applied to construct the gCM. Referring to the Compose transformation in Figure 35 page 67, the computation is simple. Starting at the root, the edges of the Composed-of relation are extracted, using the relation's transitive closure iterator method, and are inserted into the configuration. When the iteration terminates, the gCM is complete. Using the gCM, the Refine transformation iterates over the components, applying the initial constraints found in the RG. When the iterator completes, the specific component model is complete. Because there are no object-specific constraints, it is not possible to add new constraints during the transaction. Therefore, monotonic logic is sufficient, and the calls to Satisfiable and CheckConsistency methods become no-ops.

Case 2. This case expands on the previous one by allowing object-specific constraints. Object-specific constraints affect computations within the Refine transformation. Because the constraints associated with a specific selection may affect other selections, interdependencies may exist between individual refinements. As a consequence, it is not possible to perform all of the refinements in parallel.

Case 2 uses the same code as in case 1, depicted in Figure 37. The arguments to select-basic are the same, and the Compose transformation performs the same computations producing the same gCM. The difference between the two cases is found in the computations carried out within the Refine transformation. Refer to the code for "CASE 1" in the Refine transformation in Figure 36 (on page 69) starting with the line:

```
    if not(obj.Constraints.Empty())
```

Because this test will yield TRUE if object-specific constraints exist, the computations described there are performed. The constraints associated with the selected object are added to the constraint set of the configuration. Then, the constraint set is checked for satisfiability, and the consistency of previous selections is checked for consistency against the new constraint set.

Object-specific constraints come in two flavors: those that affect only subsequent refinements (monotonic) and those that may affect all refinements (non-monotonic). For object-specific constraints that affect only subsequent refinements, monotonic logic is

```
Basic_Component_Selection(RG,Rel)
    RG  : RequestGraph,
    Rel : Relation;
  {
      select-basic(RG,"component",Rel,RG.Config.Root,
                   Rel.Closure(RG.Config.Root));
  }

select-basic(RG,Model,Relation,Node,Iterator)
    RG       : RequestGraph,
    Model    : string,
    Relation : Relation,
    Node     : Composite,
    Iterator : Iterator;
  {
      Compose(RG,Model,Relation,Iterator);
      Refine(RG,Model,Relation,Node);
  }
```

Figure 37: The Basic Component Selection Process

sufficient. For object-specific constraints that affect all refinements, non-monotonic logic must be used. When monotonic logic is used, backtracking is unnecessary because previous selections can not be invalidated. As a consequence, CheckConsistency method becomes a no-op. Conversely, non-monotonic logic demands backtracking on failure to the point at which an assumption was made that is contradicted by the new constraint.

Case 3. When selection-specific models are supported, the shape of the component model is a function of the particular objects selected during the composition process. To construct selection-specific models, the Composed-of relation used to described the model must be dynamically constructed. This implies that the applications of the Refine and Compose transformations must be intertwined.

Figure 38 shows a program fragment that captures the dynamic component selection process. In Dynamic_Component_Selection, the relation parameter is bound to a newly constructed empty relation, after which Select-dynamic is then called and refines the root component. If the root can be refined, then its Depends attribute is used to augment the empty relation. Because at this point there are no other nodes in the component model, the Refine transformation terminates and the Compose transformation starts. With the tuples added to the relation during the previous Refine transformation, it is now possible to add nodes and edges to the component model. When there are no more nodes and edges to be added, i.e., the iterator terminates, the Compose transformation terminates.

At this point, select-dynamic is recursively called and the process starts all over again. Refine refines the nodes added to the model by the previous Compose and uses their Depends attributes to augment the relation. In turn, Compose uses the new edges to augment the component model. This process continues to recurse until there are no new relation entries, and there are no more components to refine.

Case 4. Case 4 is a union of the complexities found in cases 2 and 3. When both object-specific constraints and selection-specific models are permitted, the component selection process can become unwieldy. Complex interactions can occur between the various dependencies and constraints causing severe backtracking.

5.3.4 Operator Selection

As described in Section 2.2.1.2, the process for creating operator models is similar to the component selection process. In fact, the same four cases described for component selection in the previous section also apply to operator selection. The select-basic sub-procedure in Figure 37 and select-dynamic sub-procedure in Figure 38 are also used in operator selection.

The basic and dynamic operator selection processes differ from component selection only in their bindings. The code fragments for both are found in Figure 39. Notice that the iterator for Basic_Operator_Selection is the FindPath method of the operator graph defined in Section 4.2.5. FindPath searches through the operator graph, finds a legal sequence between its two arguments and produces a stream of OG edges. The Compose transformation uses these edges to construct the gOM. The Refine transformation

```
Dynamic_Component_Selection(RG)
    RG        : RequestGraph,
  {
      Composed-of.New(Rel),
      select-dynamic(RG,"component",Rel,RG.Config.Root,
                     Rel.Closure(RG.Config.Root));
  }

select-dynamic(RG,Model,REL,Node,Iterator)
    RG        : RequestGraph,
    Model     : string,
    RREL      : Relation,
    Node      : Composite,
    Iterator  : Iterator;
  {
      Refine(RG,Model,REL,Node);
      if (Compose(RG,Model,REL,Iterator) = 0)
         select-dynamic(RG,Model,REL,
                        RG.Config.BFS(Model,Node),
                        Iterator);
  }
```

Figure 38: The Dynamic Component Selection Process

```
Basic_Operator_Selection(RG,OG)
    RG  : RequestGraph,
    OG  : OperatorGraph;
  {
     select-basic(RG,"operator",OG,
        OG.FindPath(RG.Root,RG.Product));
  }

Dynamic_Operator_Selection(RG)
    RG  : RequestGraph;
  {
      OperatorGraph.New(OG);
      select-dynamic(RG,"operator",OG,
          OG.FindPath(RG.Config.Root,RG.Config.Product));
  }
```

Figure 39: The Operator Selection Processes

is then applied to the gOM, producing an sOM. For Dynamic_Operator_Selection, the operator model is a function of operator selections. Therefore, an empty OG is used to start the composition process and is dynamically constructed as refined operators are selected.

5.3.5 The Merge Transformation

The configuration construction process is comprised of the application of the Merge transformation to the RG (after both the component and operator selection processes have successfully completed). The Merge transformation augments the configuration, by filling in all the information that can be statically inferred from the specific component and operator models.

Figure 40 depicts the Merge Transformation. The formal parameters for Merge are a RequestGraph, an OperatorGraph, a Node, and an Action string. The node describes the location in the configuration from which to start the merge. The Action string defines two cases: Case 1 when the Action argument is LABEL, and Case 2 when the Action argument is EXPAND.

Case 1: Calling the Merge transformation with LABEL, implies that a simple OG is being used, as in systems like Make[28]. Simple operator graphs support simple types and derives edges between them. These are discussed in greater detail in Section 6.1. When simple OGs are used, there is no distinction made between the Derives and Composed-of relations within a configuration. They are subsumed by the Depends-on relation. As a consequence, instead of the configuration supporting two models, one for components and one for operators, only one model exists. To complete the configuration, operators must be inserted into the configuration (between model edges).

In Case 1, Merge transforms the configuration by looking up operators in the OG that correspond to edges in the model and inserting them into the configuration. This is accomplished by a loop that iterates over the transitive closure of the Depends-on relation producing a stream of edges. For each edge in this stream, the FindOperator method of the OperatorGraph is called to find the appropriate operator. The FindOperator method uses the edge to find a corresponding edge in the OG. The operator associated with this edge is then returned. The operator is then used to label the edge in the configuration. This is done by inserting the operator between the domain and co-domain of the original edge.

To clarify this concept, a simple example is described. Assume that the following knowledge exists in the database:

```
depends-on(test.exe,test.o)
depends-on(test.o,test.c)
og-edge(o,exe,ld)
og-edge(c,o,cc)
```

Before the call to the Merge transformation, the configuration contains only one model which looks like:

$$\text{test.c} \rightarrow \text{test.o} \rightarrow \text{test.exe}$$

```
Merge(RG : RequestGraph, OG : OperatorGraph,
    Node :  Composite, Action :  string )
{
    if (Action = LABEL) /* CASE 1:  Make-like Systems */
    {   iterate (edge = RG.Config.Depends-on.Closure(Node) )
        {   operator = OG.FindOperator(edge)
            RG.Config.OperatorInsert(operator,edge);
        }   }
    else if (Action = EXPAND) /* CASE 2:  */
    {   op = Node;
        iterate(Config-Node = RG.Config.NextNode("operator",op))
        {   if ((op.Kind="aggregate") and not(op.Delay))
            {
                op = Config-Node.node;
                RequestGraph.New(TEMPLATE);
                TEMPLATE.Config.SetProduct(op.DeriveTo);
                TEMPLATE.CS = RG.CS;
                iterate (from = operator.Inputs.Next() )
                {   RequestGraph.New(OM);
                    OM = TEMPLATE; /* copy */
                    OM.Config.SetRoot(from);
                    Compose(OM,"operator",OG,
                        OG.FindPath(OM.Config.Root,
                            OM.Config.Product))
                    Refine(OM,"operator",nil,OM.Config.Root)
                    RG.Union(OM);
                }
            }
        }
    } /* end expand*/
} /* end Merge */
```

Figure 40: The Merge Graph Transformation

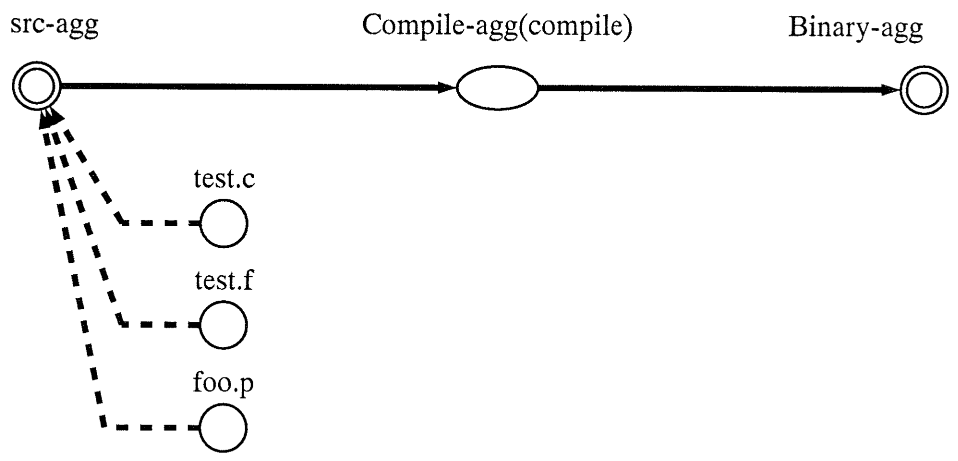

Figure 41: A Partial Configuration Before Merge.

After the call, the configuration is augmented with operators and looks like:

test.c $\xrightarrow{cc}$ test.o $\xrightarrow{ld}$ test.exe

Case 2: Case 2 implies the use of a sophisticated OG which supports the concept of aggregate operators. Recall that an aggregate operator provides an attribute that specifies a product to be produced for each member in the input collection. It is the responsibility of the Merge transformation to use this information to fill in the configuration.

When the input to an aggregate operator is the result of a preceding input-sensitive operator, the members of the input set are not statically known, and therefore the Merge must be delayed until runtime (when the information is available). This case is explained in greater detail in Section 5.4.2 when the Build transformation is discussed.

In Merge, when an aggregate operator is encountered in which the input set is statically known, there exists sufficient information to fill in incomplete parts of the configuration. Figure 41 shows an example of a configuration with an aggregate operator before the Merge. Figure 42 shows the same example after the application of the Merge transformation. Notice the new nodes and edges emanating from objects that are members of the input to the aggregate operator.

To perform aggregate operator expansion, the following computations (specified in the transformation in Figure 40) are performed. The outside loop starting at Node, iterates through all of the operators in the configuration. This iterator produces a stream of operators. Within the loop, the following happens: for each operator, the operator is inspected to determine whether it is an aggregate. If the operator is not an aggregate, then the rest of the code in the loop is not executed. If the operator is an aggregate, but is the result of a prior input-sensitive operator that has not yet executed, then the Delay attribute of the operator will be set to TRUE. This implies that there is insufficient

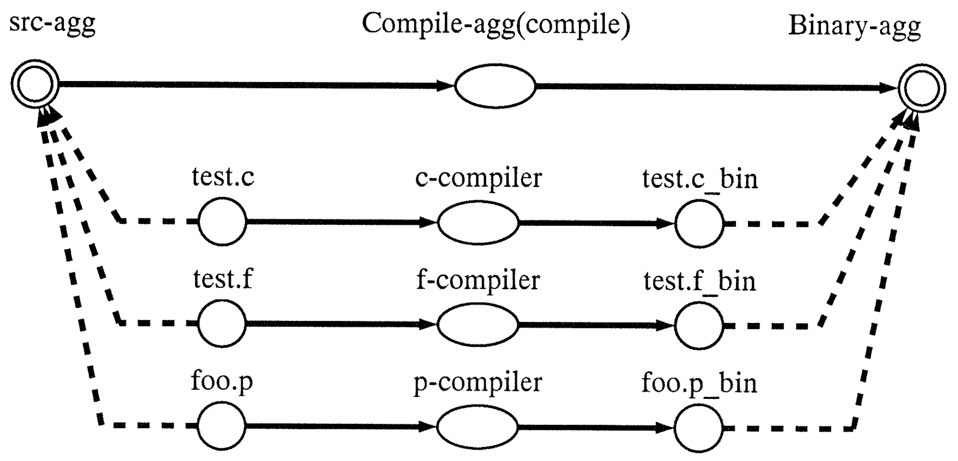

Figure 42: A Partial Configuration After Merge.

information at the moment to carry out the transformation. The merge is delayed until after the information becomes available.

When a non-delayed (Delay=FALSE) aggregate operator is encountered, the Merge transformation has sufficient information to fill in the missing nodes and edges. The inner iterator produces a stream of the elements in the input set. For each element, the element type and desired output type are used to construct a request that is then used to compose a sub-configuration. This code can be viewed as a recursive call to the composition process, followed by a grafting of the sub-configuration into the original configuration.

Before the inner iterator loops over the input set, a template for the request is set up. This involves creating a new instance of an RG and initializing it. The product attribute in the configuration is set to the DeriveTo attribute of the aggregate operator, and the current configuration constraint set are copied into the template.

Within the body of the inner iterator, the following occurs for each element ("from"). A new instance of an RG is created called the operator model (OM in the code). Then the template is copied to the OM, and the root of the configuration is set to "from". The OM now looks exactly like the translation of a user request. This OM is now used to compose a generic operator model, and then refined into a specific operator model. This is done by calling the Compose transformation to compose the gOM, and then the Refine transformation to produce the sOM. This sequence of transformations is exactly the same as the one used in the Basic_Operator_Selection procedure found in Section 5.3.4. Finally, after the sOM is complete, the newly constructed RG is grafted back into the original configuration at the input set element.

After iterating through all of the inputs of the aggregate operator, the outer iterator moves on to the next operator in the configuration. The same inner loop is then applied to any non-delayed aggregate operator found. The outer iterator terminates when all of the operators in the configuration have been visited.

There are several interesting variants that can occur within the Merge transformation. The first is when the Product specified by the aggregate operator involves a very complex request. This request could involve the inclusion of many operators for each element. Second, any or all of the operators added to the configuration as a consequence of previous Merges could also be non-delayed aggregate operators, in which case each, in turn, must be merged. Third, the elements of the aggregate could have more than one type. In this case, as long as there is a path from the type of each element to the DeriveTo operator, Merge will not fail. On the other hand, if the input aggregate is a set (has a single element type) then it is possible to push the computation of the gOM (the Compose application) outside the inner iterate loop, thus reducing redundant computations. Another variant involves the substitution of the Basic_Operator_Selection code with the code for Dynamic_Operator_Selection. This variant is extremely complex because backtracking could cause the entire transaction to be restarted.

5.4 Manufacture

This section details the manufacturing process depicted in Figure 32. The manufacturing process is comprised of two transformations: Schedule and Build. The Schedule transformation discussed in Section 5.4.1 uses as input the configuration produced by the composition process and produces an operator schedule. Using the operator schedule, the Build transformation detailed in Section 5.4.2, performs the necessary operator applications. As operators are applied other transformations that modify either the configuration, the operator schedule or both may have to be applied, before manufacturing can proceed.

5.4.1 The Schedule Transformation

The Schedule transformation modifies the RG by instantiating the value of its OpSched attribute. It uses the configuration contained in the RG, and a consistency predicate, to statically determine what operators need to be manufactured. The dynamic (or planning) aspects of scheduling that arise from input-sensitive operators and complex predicates are handled within the OpSched object. As the statically constructed schedule is executed in the Build transformation (discussed in Section 5.4.2), operators may be added or deleted, and the entire schedule may have to be re-evaluated.

In order to cope with the complexities introduced into the SCM process by complex predicates and input-sensitive operators, methods for dynamic manipulation of the operator schedule are needed. Dynamic scheduling is similar to the concept of dynamic planning used in the area of artificial intelligence. Lesser and Huff have even applied this concept to try and model part of the software development process[35]. Think of the schedule as a plan of action. As the plan executes, new information becomes available that causes the original plan to be modified.

When complex predicates are supported, additional information must be derived from changed objects. This information is then used to revise the schedule. An example

```
define OperatorSchedule
   description: "Working Graph for GT  request satisfaction"
   attributes:
     Ops          : relation(Operator,Operator);
     ReadyList    : listof(Operator)
     Predicate    : Predicate;
     IsParallel   : boolean;
   methods:
     boolean   Apply(op: Operator  wait : boolean);
     boolean   GetMachine();
     boolean   ReSchedule();
     boolean   InsertOp(Operator);
     boolean   DeleteOp(Operator);
     iterator  Operator Next();
end;
```

Figure 43: The Operator Schedule Class Specification

of the smart-recompilation predicate was discussed in Section 3.4.1, where it was shown how operators were removed from the schedule based on the information produced by intermediate processing steps.

When input-sensitive operators are supported, the configuration will be augmented as new information becomes available. This augmentation process, discussed in Section 5.4.2, adds new operators to the configuration. The addition of new operators to a configuration implies that the operator schedule must be updated.

Before discussing the details of Schedule transformation, the OperatorSchedule class needs to be defined. Specified in Figure 43, the OperatorSchedule class encapsulates the concept of a scheduler. Scheduling is well understood[30]. Basic scheduling algorithms can be found in virtually any introductory book on operating systems design, and in fact, there are at least two SCM systems today, DSEE[46] and Parmake[5], that use scheduling algorithms to perform parallel builds. The following paragraphs explain the attributes and methods of the OperatorSchedule Class.

Ops. The Ops attribute encapsulates dependencies between the operators that must be manufactured. Edges are inserted into this relation by the InsertOp method. The Ops attribute is used to determine when to put operators on the ReadyList.

ReadyList. The ReadyList attribute contains the list of operators that can be executed. This list is initially constructed statically by the Schedule transformation. As operators are executed, other operators found in the Ops attribute, that depend on their output will be updated to reflect the state of readiness of their input. When all of their inputs are available, they are added to the ReadyList.

Predicate. The value of the Predicate attribute is used in the ReSchedule method.

If the value is undefined, then ReSchedule uses the time-stamp predicate.

IsParallel. The IsParallel attribute is set TRUE if the number of machines is greater than 1. It is set in the Schedule transformation.

Apply. The Apply method is called to invoke an operator. It sets up an environment for operator application, starts a process and executes the operator. If the `wait` argument is TRUE, then Apply waits for the termination of the process. If the `wait` argument is FALSE, it immediately returns control to the caller. An interrupt handler manages process termination and machine de-allocation.

GetMachine. The GetMachine method acts as a resource allocator for machines. If there is a machine available in the pool, the machine is marked allocated and control returns. When the pool is empty, GetMachine waits for one to become available. Machines are de-allocated when operator applications complete.

ReSchedule. The ReSchedule method used in the Build transformation is discussed in the next section. It applies the predicate found in the Predicate attribute to the current OpSched object and produces a new one.

InsertOp. The InsertOp method adds operator edges to the Ops attribute. It inspects the Inputs and Output attributes of the operator and adds the appropriate edges.

DeleteOp. The DeleteOp method is used by the ReSchedule method to remove operators from the schedule. When the information needed to compute a complex predicate becomes available, ReSchedule is called to re-evaluate the schedule. During this computation, it is possible that based on the new information, subsequent operators can be removed.

Next. The Next method gets the next operator on the ready list. If the ready list is empty, then the Next method waits. The interrupt handler, upon completion of an operator application, inspects the Ops attribute and marks the operator as completed. It then updates the operators that depended on the completed operator to reflect that one of their inputs is now ready. If at this time, all of the inputs to an operator are available, the operator is added to the ready list.

Now that the operator schedule has been explained, it is possible to discuss the details of the Schedule transformation shown in Figure 44. At an abstract level, a complete configuration can be thought of as a complete operator schedule—it contains all of the operators and derivational interconnections needed to construct the product. If a configuration has never been manufactured before then the operator schedule produced by the Schedule transformation will look very similar to the information contained in the configuration. On the other hand, if a configuration has been previously manufactured, the complete schedule found in the configuration may contain many unnecessary steps, and the Schedule transformation must prune the schedule of unnecessary operators. This is done by applying the simple consistency predicate (`simple-consistency-predicate`) discussed in Section 2.2.2.3 to the operators in the configuration. If the application of the predicate yields TRUE, then the operator is consistent and does not need to be (re-)manufactured. If the predicate yields FALSE, the operator is inserted into the operator

```
Schedule (
    RG : RequestGraph,
    Predicate :  Predicate,
    machine-count :  int )
    {   RG.OpSched.SetPredicate(Predicate);
        RG.OpSched.SetIsParallel( machine-count > 1 ?
                    TRUE : FALSE );
        iterate ( op = RG.Config.O-Nodes.Next())
            {   if ( not(simple-consistency-predicate(op))
                    RG.OpSched.InsertOp(op)
            } }
```

Figure 44: The Schedule Graph Transformation

schedule.

The formal parameters to the Schedule transformation are an RG, a Predicate and the number of machines available for manufacturing. The computations performed by this transformation are quite simple. First, the Predicate attribute of the OpSched is set to the Predicate argument. The Predicate argument will have a value only when a complex predicate is used. The Predicate attribute is used in the ReSchedule method to re-compute the schedule as new information becomes available. Then, the IsParallel attribute is set, denoting if there is more than one machine available for manufacturing. Finally, the iterator produces a stream containing all of the operators in the configuration. For each operator, it applies the consistency predicate. If the predicate evaluates to FALSE, the operator is inserted into the schedule.

5.4.2 The Build Transformation

The Build transformation is used to apply only those operators that need to be executed within a configuration to produce the desired product. Figure 45 depicts this transformation. The formal parameters to Build are an RG, an Operator, and the number of machines available for carrying out operator application. The RG contains most of the information needed to carry out this process. The Operator describes the location within the operator schedule from which to start the computations. The machine-count parameter is an integer describing the number of machines available.

There are four cases of interest in the Build Transformation. These four cases are based on whether or not complex predicates and/or input-sensitive operators exist, and on whether more than one machine is available.

Case 1. In the simplest case, there are no complex predicates, no input-sensitive operators and only one machine available. As a consequence, the operator schedule, once statically determined, is unmodifiable. Because there is only one machine, a linear ordering is imposed on the operator schedule. For this case, only the last three lines of Figure 45 would be executed. First, the Apply method of the operator schedule would be called with a TRUE argument value. The wait argument to Apply states that the operator application should terminate before the method returns control to the invocation site. Once the operator is applied, the OpSched object's Next method is called to yield the next operator to be manufactured. Build then recursively calls itself with the new operator. This recursive cycle continues until there are no more operators to apply (or until an operator application fails).

Case 2. This case is similar to case 1 except that there is more than one machine available for manufacturing. When there is more than one machine available, the last four lines of the Build transformation are executed. Build calls the GetMachine method of the operator schedule to allocate a machine. GetMachine will return only when a machine is available. Build then calls the Apply method with a FALSE argument, signifying the method to return immediately. The Next method is then called to grab the next operator to be manufactured. Next waits if there are no operators on the ready list. Finally, the Build transformation is recursively called to manufacture the remaining operators in the

```
Build (
    RG : RequestGraph, /* the request graph */
    Operator :  Operator, /* current operator to execute */
    machine-count :  int)
    {
        If (Operator.GetDelay() = TRUE)
        { /* CASE 3 */
            RG.OpSched.Apply(Operator,wait);
            Operator.SetDelay(FALSE);
            iterate ($\forall$ ops $\in$ RG.OpSched.Ops
                [ Operator.Output $\in$ ops.Inputs])
            {
                Merge(RG,OG,ops,EXPAND)
            }
            RG.OpSched.ReSchedule();
            RG.OpSched.Next(Operator);
            Build(RG,Operator,machine-count);
        }
        else if (not(Rg.OpSched.Predicate = nil))
        { /* CASE 4 */
            RG.OpSched.Apply(Operator,wait);
            RG.OpSched.ReSchedule();
            RG.OpSched.Next(Operator);
            Build(RG,Operator,machine-count);
        }
        else
        { /* CASES 1 and 2 */
            if (machine-count>1) RG.OpSched.GetMachine();
            RG.OpSched.Apply(Operator,RG.OpSched.IsParallel);
            RG.OpSched.Next(Operator);
            Build(RG,Operator,machine-count);
        }
    }
```

Figure 45: The Build Transformation

schedule.

Case 3. For this case the operator argument is input-sensitive. Before any other operators can be scheduled for manufacture, the input-sensitive operator must be manufactured and its results used to fill in missing information in the configuration. This is done by calling the Merge transformation. Recall that the Merge transformation, when "Action = EXPAND" adds nodes and edges to configurations.

Case 4. Case 4 deals with complex predicates. As stated in the previous section on the Schedule transformation, complex-predicates may cause the removal of operators from the statically constructed operator schedule. To handle this case, the Apply method of the OpSched object is called serially. When the operation is complete, the schedule must be re-evaluated. This is done by walking the operator schedule and applying the predicate along the way. If the predicate finds an operator that yields TRUE, then the state of the operator and its derived output are set to consistent. As subsequent operators in the schedule are checked for consistency (before the predicate is applied) if all of their inputs are now in a consistent state, then the operator state can be changed to consistent, and the predicate need not be applied.

5.5 Summary

This chapter presented the Graph Transform (GT) model for the software configuration management process. Described as a sequence of transformations on directed acyclic graphs, the GT model is able to explain both the basic SCM process described in Chapter 2 and the complexities introduced in Chapter 3. The model details the interactions between the transformations and their inherent and mutually recursive nature. In the next chapter, the usefulness of the GT model is validated by describing existing SCM systems as instances of the model.

CHAPTER 6

MODEL VALIDATION

In this chapter the four prominent SCM systems, Make, DSEE, Adele, and Odin are described as instances of the GT model. Describing these systems as model instances serves two purposes. The first is that it validates the claim that the GT model is capable of capturing the essential, yet diverse functionality of existing systems. Secondly, because the GT model provides a unifying framework for description, it is possible to compare and contrast the systems in a way which is understandable to people unfamiliar with their particulars.

Make was chosen because it is a simple SCM system that most people are familiar with. DSEE provides generic models, a simple constraint language for describing how to select instances from restricted component families and the ability to perform parallel builds. Adele supports user-definable types, object-specific constraints and selection-specific models. Odin was chosen because of the complex manufacturing process it supports, its sophisticated specification language and its associated Derivation Graph.

The first four sections of this chapter explain the Make, DSEE, Adele and Odin systems, respectively, as GT model instances. This is done by describing the semantic entities each supports and then providing a GT process program that manipulates them. The internal organization of these four sections is as follows. First, a brief overview of each system is given. The semantic entities supported by the system are then discussed. These entities are grouped into four classes: types, relations, an operator graph and constraints. The process each system supports is then described as a GT process program. Whenever possible, quotes from papers or manuals are included to detail concepts supported by the particular system and to substantiate or validate concepts supported in the GT model. After describing the process, the section terminates with a discussion about the features and limitations of the system being described. Finally, Section 6.5 provides a summary of the four systems.

6.1 Make

Make [28] is the grandfather of SCM systems. It was the first system to perform automatic consistency management based on machine readable dependency relations. It is an eloquent, simple program that does a satisfactory job on small systems supporting few versions, simple tools, and only a small number of people working together.

Make and make-like systems [27,37,29,5,57,20] derive their influence from their ubiquity. Make comes free with virtually every Unix installation. Make and its variants are relatively easy to use, embody basic domain knowledge about their operating environment (directories, time-stamps, standard compilers, archive formats, etc.), and provide a standardized, parameterized, specification paradigm. A Make specification called a

Makefile is a textual representation that can be transported to a variety of hardware (and software) systems. Because Make has no direct dependence on the file system, operating system or hardware, and because version management is not integrated, Make is the closest thing to an inter-operable configuration manager available today.

On the other hand, for the same reasons listed above, Make will be forever limited. The lack of attributed object support is perhaps the single most important deficiency in Make. Without attributed object support, it is not possible to support either primitive or derived component families. However, until such time that a standardized software configuration management interface to/for an object repository comes about, Make will continue to dominate how people think about configuration control and software management on Unix and other similar systems.

6.1.1 Semantic Entities

In Make, the semantic entities are encapsulated in a Makefile, which contains static dependency information and simple transformation rules. The Dependency information is analogous to tuples in a static Depends-on relation. Transformation rules can be thought of as simple operators connected by a simple operator graph. In Make, there are no user-specifiable constraints and no way to specify complex objects, aggregate objects[1]. or object attributes.

The following is a summary of the SCM semantic entities of interest for Make, which are discussed in the paragraphs below.

- **Types:** Suffixes.
- **Relations:** Static Depends-on.
- **Operator Graph:** Simple.
- **Constraints:** None.

Types. Make provides no facilities for type construction. As a consequence, there is no aggregate or composite type support. The types known to Make are the set of file suffixes referenced in the Makefile—the file "foo.c" is of type "c". Make distinguishes between primitive and derived components by how the components are used (not by their definition). If Make does not know how to derive a suffix—there is no transformation rule to produce it—then the suffix represents a primitive type. All other suffixes are considered to be derived types. Make has no explicit concept of operators, although its transformation rules indirectly define them. Because types cannot be constructed in Make (i.e., it is not possible to associate attributes with types), the concept of object families cannot be supported. The only attributes available are those that the filesystem supports (name, suffix, creation time and creation date).

Relations. Make supports a statically-specified Depends-on relation. Dependency lines are found in the Makefile. They describe a target and the objects that target depends on. For example, in the Makefile fragment below, the "compiler.exe" object depends on the "compiler.o" object which in turn depends on the "compiler.c" and the

[1]Archive libraries are a specialized case of an aggregate type supported by Make even though there are no generalized facilites for aggregate construction (or any other kind of type construction).

"compiler.h" object.

```
        compiler.exe: compiler.o
        compiler.o:   compiler.c compiler.h
```

Operator Graph. Make supports simple transformation rules. A transformation rule is characterized by a pair of suffixes followed by a sequence of commands. The suffixes denote the input and output types for which this transformation is applicable. The sequence of commands represent the "tool" to be applied. The suffix transformation rule below states that a file whose suffix is ".c" can be transformed into a file of the same name whose suffix is ".o" by applying the "cc" tool.

```
        .c.o:
            cc $<
```

By representing the transformation rules as edges in a binary relation, an OG can be constructed over the suffixes. This OG represents all of the legal sequences of type transformations that the Makefile supports. The OG is relatively simple. The implicit suffix transformations are modeled as simple operators—one suffix in, one suffix out. Modeling explicit rules is slightly more difficult. Explicit rules in make can be thought of as the components that make up a limited form of selection-specific operator models[2]. Take for example the following explicit transformation rule:

```
        test.o : test.c test.h
                 cc -special-options test.c
```

This rule states that when a "test.o" object is selected, the specific transformation should be applied instead of the general one. Basically there are two ways to handle this. One way is to to create a new Derived-Component, say, "special-test-o" type and a special operator for it. Then, by adding the appropriate super-type relation edge (`super-type(o,special-test-o)`) the object produced by the transformation can be used in an subsequent transformations that calls for an object with an ".o" suffix. A better way is to associate with the specific component a component-operator constraint that defines the specific tool (or tool version/variant/parameter(s)) needed to transform this object. Then, when the specific object is selected the component-operator constraints would be added to the constraint set.

In Make there is no support for aggregate, composite or input-sensitive operators in Make. The OG does not support equivalence or super-type relations because there are no facilities in Make to specify them.

In Make it is possible to perform macro substitutions within transformations to propagate tool options. Unfortunately, these substitutions are purely textual and therefore cannot be used to make distinctions between derived objects.

Constraints. Make provides no mechanisms for describing or using constraints.

[2]Make is the only one of the four systems that supports any form of selection-specific operator model.

```
Make(Target,Flags,Depends-on,OG) {
   RequestGraph.New(RG);
   RG.Config.SetProduct(Target);
   RG.CS.Constraints.Add(Flags);
   Compose(RG,"component",Depends-on,
           Depends-on.Closure(RG.Config.Product));
   Merge(RG,OG,RG.Config.Product,LABEL);
   Refine(RG,"operator",nil,RG.Config.Product);
   Schedule(RG,nil,1);
   Build(RG,RG.OpSched.ReadyList.First,1);
}
```

Figure 46: The Make-GT Process Program

6.1.2 The Process

In Make the SCM process is quite simple. The dependency graph for the target is constructed from the Makefile. The nodes in the graph are then visited in a depth-first fashion. As the node is visited, a simple time-stamp predicate is applied to determine if the object needs to be (re-)constructed. The following summarizes the processing capabilities Make supports, which are then discussed below in greater detail.

(1) **Object-Specific Constraints:** Not Supported.
(2) **Selection-Specific Models:** Limited form for operators.
(3) **Input-Sensitive Operators:** Not Supported.
(4) **Complex Predicates:** Not Supported.

The Process Program

Composition. The Make process program depicted in Figure 46 is an abstract representation of the SCM process that Make provides. In the real Make, all processing is done in one pass over the dependency graph.

The process depicted in Figure 46 begins by creating a new RG instance and setting the Product attribute to the Target (the target in Make is specified as an argument when Make is invoked). The Flags argument represents the operator flags or options that are textually substituted into the operators. After setting up the RG, the process proceeds to compose the component model using the Compose transformation. The Compose transformation uses the statically constructed instance of the Depends-on relation to extract dependency information and construct the component model in the configuration. Compose applies the Closure method of the relation to extract the entire transitive closure of the target. Next, the Merge transformation is applied with "action=LABEL". Merge walks over the depends-on edges in the component model. For each edge, it finds the matching edge in the OG and inserts the appropriate operator into the configuration, thus constructing the operator model. The Refine transformation is then called to instantiate the attributes of the operator model by propagating the Flags (found in the constraint set) to the appropriate operators.

Manufacture. Once the configuration is complete, the manufacturing process begins. The Schedule transformation is called to construct the operator schedule. The arguments to the Schedule transformation specify that no complex predicate is used, and that there is only one machine available for operator application. Because the configuration is complete (there are no complex predicates or input-sensitive operators), a complete operator schedule can be statically constructed. The Schedule transformation applies a simple consistency predicate (time-stamp) to all the operators in the configuration. If the result of the application of the predicate is FALSE, the operator is placed in the schedule. Once the operator schedule is constructed, the Build transformation is applied. In the Build transformation, the code corresponding to Case 1 of Section 5.4.2 (no complex-predicates, no input-sensitive operators, 1 machine) is executed. The operators in the schedule are applied, one-by-one, in a linear order. When the last operator is applied, the target is made available.

An excerpt from Feldman's seminal paper on Make describes the basic process:

> The basic operation of Make is to update a target file by ensuring that all of the files on which it depends exist and are up to date, then creating the target if it has not been modified since its dependents were. Make does a depth-first search of the graph of dependencies. The operation of the command depends on the ability to find the date and time that a file was last modified[28].

Discussion. Make, because of its simplicity, is able to perform both the composition and the manufacturing processes in one pass over the dependency graph. By inspecting the process program above, several things should become apparent.

Configurations are never really constructed in Make. The text in the Makefile is converted into an internal graph representation and then walked in a depth-first, left-to-right fashion. After the computation completes, the constructed graph is thrown away. The next time the same request is made, the graph has to be re-computed. One simple way to improve Make's efficiency is to store the Makefile in an internal representation and only recompute it whenever the Makefile is changed.

Due to the lack of attributed objects, it is not possible in Make to distinguish between objects derived with the same compiler but with different Flags (options). Although the Flags are propagated to the appropriate place and then used to satisfy the request, this information is not recorded or maintained. Consequently, if the same target is requested but the Flags are changed, the Make process would determine that all the objects were consistent, an incorrect conclusion.

6.2 DSEE

DSEE [46,47,48,49,39] is s modern, commercially available SCM system. It provides facilities for generic models, integrated version management and selection, parallel building and derived object management support.

One distinguishing characteristic of DSEE is its concept of the separation of version specification from component definition. DSEE was one of the first systems to support generic system models, configuration threads and bound configurations. System models (SM), represented as text files, contain information similar to the generic component model described in Section 2.2.1.1. An SM defines a static structure that describes the generic components and the structural interconnections needed to produce a software system. DSEE defines Configuration threads (CT) as rule based descriptions (constraints) that describe the versions of components to use for a particular build. When a CT is applied to a SM (in GT terms, when a SM is refined using the CT), the generic objects in the SM are instantiated (refined) into concrete components. The resultant object created by CT application is called a bound configuration thread (BCT), which is similar to the specific component model defined in Section 2.2.1.1.

Files in the Domain/OS[3], the operating system supporting DSEE, are capability based and the "file system offers ordinary, unmodified programs transparent access to any version of a DSEE element via the Domain system's IOS stream facility[39]". The Domain/OS also supports network-wide virtual address space, transparent remote file access, remote paging, a distributed database management system, reliable, immutable files, and server processes. Because version management is built-in, the filesystem provides a version attribute for files. The version attribute is used to define generic components (or component families). Configuration threads are used to refine generic components into concrete ones by instantiating their version attributes.

6.2.1 Semantic Entities

DSEE's semantic entities are encapsulated in two types of objects: system models and configuration threads. The SM describes the generic components and the translation rules needed to build a software system. The CT contains the version selection constraints. The following is a summary of the DSEE semantic entities, which are discussed in the paragraphs below.

- **Types:** Suffixes, aggregates.
- **Relations:** Static Composed-of.
- **Operator Graph:** Simple
- **Constraints:** Version selection predicates.

Types. DSEE supports primitive and derived components, both of which can be either simple or aggregate. Simple types in DSEE, as in Make, are defined by the file suffixes present in the system model. Unlike Make, DSEE has an "aggregate" key word in its system model specification language for constructing aggregates. An example of aggregates can be seen in the SM fragment shown in Figure 47.

Component elements reside in the Domain/OS filesystem. In addition to name, type and creation date, the filesystem supplies a version attribute for its files. The version attribute is used to describe a limited form of object families. The concept of object families is limited because it is not possible for users to define additional attributes.

[3]Domain/OS is a trademark of the Apollo Corporation.

```
model generic_compiler =
    aggregate FE =
        element lexer-Spec =
            depends_source
                ...
        end;
        element parse-spec =
            ...
    end;
    aggregate Translator =
        ...
end;
```

Figure 47: DSEE System Model Fragment

Versions and version management are handled internally by the Domain/OS. It is not necessary to check out an object in order to send it to a tool. Because the OS is capability based, it will provide the appropriate stream to any tool in the system when that tool opens a file:

> This enables the Display Manager, shells and ordinary applications (like compilers and text formatters) to read any version of a source element directly from a library. No special help is needed to copy the version into a plain text file[39].

Relations. DSEE supports static instances of the Composed-of relation. A relation instance is constructed when a system model is compiled. The system model, a variant on a module interconnection language[45,67], specifies the elements of the Composed-of relation by their hierarchical relationships. The following excerpt from one of the DSEE manuals defines a system model.

> A system model is a sort of blueprint of the system's components, their interrelations, and their translation rules. ... The system model gives a static description of the structure of the system in terms of its buildable components, their constituent elements, and their hierarchical relationships. The system model also gives translation rules to be used to produce the derived objects. Translation rules can include translation options, which you can decide to use or not for any given build of the system[39].

Part of a compiler example might look like the SM fragment shown in Figure 47. The Composed-of relation is constructed as the SM is compiled. For the compiler example of Chapter 2, the edges inserted into the relation would be the same as the edges depicted in the compiler model depicted in Figure 1.

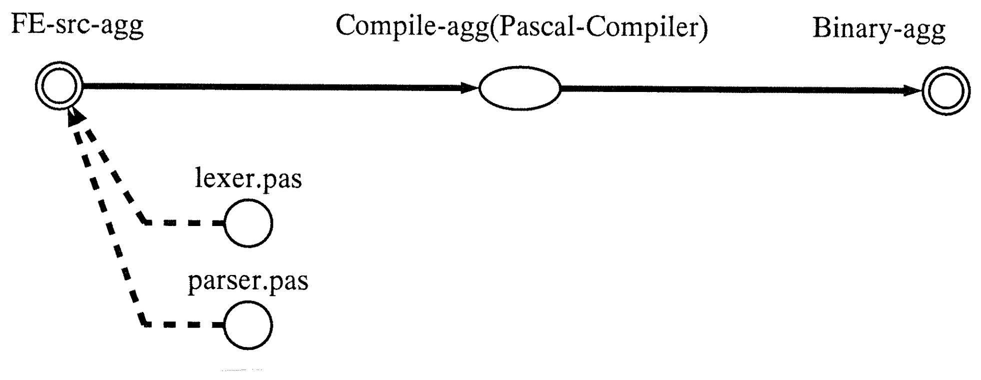

Figure 48: Constructed DSEE configuration fragment

Operator Graph. The operator graph for DSEE is both more and less sophisticated then the one found in Make. It is more sophisticated because of DSEE's ability to define aggregates. Aggregate operators (not input-sensitive) are constructed when the system model is compiled. For example, if an SM contained a specification for an aggregate called FE that contained two elements "lexer.pas" and "parser.pas", then it would be possible to construct both a set aggregate operator and its input set. As depicted in Figure 48, the aggregate set operator specifies the application of the Pascal-Compiler operator to a set of Pascal sources, producing a set of binaries. As the SM is compiled, the input set to this operator can also be statically determined because all of the names and types of its elements are contained in the SM. As a consequence, when the composition process applies the Merge transformation, a complete configuration can be constructed.

On the other hand, DSEE's OG is less sophisticated than Make's because there is no support for type inferencing. The type of a component exclusively determines which operator is used. For example, whenever a ".pas" object is encountered in the SM, the Pascal-Compiler will be applied.

Constraints. DSEE has limited support for constraints in the form of configuration threads. Configuration threads are defined in the DSEE manual as follows:

> A configuration thread specifies which versions of elements to use in the build. A configuration thread consists of an ordered list of rules. Each rule contains a predicate that identifies the elements to which it applies and a specification of the version to use for those elements[39].

There is no support for object-specific constraints, or selection specific models.

6.2.2 The Process

When a user wants to build instances of a software system using DSEE, there are three steps that must be performed. The first is to tell DSEE which system model to use. By doing so, the user instructs DSEE to take the textual representation of the system model and "compile" it into an internal representation in working memory. This representation contains a static Composed-of relation and a simple operator graph. The internal representation persists until either a new model is requested or the DSEE interpreter process terminates. Second, when the user wants to build a specific instance (version) of the system, he/she tells DSEE which configuration thread, or CT, to use. The CT is then applied to the SM producing a bound configuration thread (BCT): SM x CT $\rightarrow$ BCT. The BCT represents a complete configuration—it contains all of the components, operators and structural and derivational interconnections needed to build a specific product. Finally, the user requests a "build", at which point the manufacturing process takes over. If the user then wants to work on a different instance (version) of the same system, a command to change the CT is issued, and a new BCT is constructed from the original SM and the new CT .

The following summarizes the processing capabilities supported by DSEE. Their impact on the process is described below.

(1) **Object-specific Constraints:** Not Supported.
(2) **Selection-Specific Models:** Not Supported.
(3) **Input-Sensitive Operators:** Not Supported.
(4) **Complex Predicates:** Not Supported.

The Process Program

Composition. DSEE provides a richer set of capabilities for model composition than Make. In DSEE, generic component models are defined that describe a system's structure in terms of their generic components and interconnections. This structure is then refined, using configuration threads, to determine the actual components needed to build a particular version of a system. When the SM is compiled, a static Composed-of relation representing the generic component model is constructed. An specific operator model is also constructed from the information contained within the SM. The translation rules, translation parameters and aggregate definitions are all used in the sOM construction. To be consistent with the GT model, this constructed sOM will reside in an instance of the OperatorGraph class. The constructed sOM is not exactly an OG. This is because it pertains only to the specifics in one system model, has no inferencing capabilities, and the operators contained therein are instantiated with parameters specified in the SM, instead of being applied on a per request basis.

Figure 49 depicts the GT process program for DSEE. The most significant difference between this process program and the one for Make deals with the construction of the component model. As described below, a generic component model is constructed first and then the Refine transformation is used to transform the generic component model into an specific component model. The CT represents constraints used in the

```
DSEE(Root,CT,Machine-count,Composed-Of,OG) {
   RequestGraph.New(RG);
   RG.Config.SetRoot(Root);
   RG.CS.Constraints.Add(CT);
   Compose(RG,"component",Composed-of,
          Composed-of.Closure(Root));
   Refine(RG,"component",nil,RG.Config.Root);
   Merge(RG,OG,RG.Config.Product,EXPAND);
   Schedule(RG,nil,Machine-count);
   Build(RG,RG.OpSched.ReadyList.First,Machine-count);
 }
```

Figure 49: The DSEE-GT Process Program

Refine transformation. In Make, however, because component families are not supported, the gCM and the sCM are one and the same.

The process begins by transforming the arguments into a request graph. Then the root of the configuration and the initial constraint set are assigned values. The system model is then constructed using the Compose transformation. The Compose transformation, using a statically constructed instance of the Composed-of relation, performs the transitive closure over the relation, extracting edges and inserting them into the configuration. When the Compose transformation completes, the SM is complete. The SM is analogous to the gCM defined in Section 2.2.1.1. It contains all of the generic primitive components (family roots) and the structural interconnections needed to build the software system.

The next step in DSEE is to apply the version selection constraints found in the CT to the SM. The result is a BCT in which all version attributes have been instantiated. This is done in the GT model by applying the Refine transformation. Because neither object-specific constraints nor selection-specific models are supported, the computations within the Refine transformation are simple. Refine walks the gCM in the configuration and applies the constraint set to each node. The result of the application of the constraint set to the node is to bind a value to the node's version attribute, thus producing a concrete component. Because object-specific constraints are not supported, there is no interdependence between the individual refinements. Therefore, it is possible for all of the refinements to be done in parallel (although DSEE does not do this). When the Refine transformation terminates, a complete sCM exists.

Finally, the sCM is merged with the OG to produce a complete configuration. The Merge transformation uses the sCM and the constructed OG produced from the system model as input and results in a complete configuration. The configuration is complete at this point because input-sensitive operators are not supported. Because DSEE supports aggregates, the action argument to the Merge transformation must be

"EXPAND". As aggregate operators (such as the Compile-agg found in Figure 48) are encountered, Merge will fill in the missing details of the configuration.

Manufacture. The manufacturing process in DSEE is also more sophisticated than in Make, for two reasons.

(1) DSEE supports parallel operator executions.
(2) Because of its derived object management support, operator parameters are recorded with derived objects.

DSEE is able to perform parallel builds in part because of the facilities the underlying operating system supplies, and in part because of its scheduler. In fact, the OperatorSchedule class defined in Section 5.4.1 is patterned (partially) after the DSEE scheduler. Because of derived object support, DSEE is able to distinguish between two derived objects that differ only in the value of a parameter used to construct them (the object resulting from the compilation of foo.c with "cc" and "-debug" is a different object then that resulting from the compilation of foo.c with "cc" and "-optimize").

Once the configuration is complete, the manufacturing process begins. The Schedule transformation is called to construct the operator schedule. To exploit DSEE's parallel build capabilities, the assumption is made that the value of machine-count is greater than 1. Because the configuration is complete and there are no complex predicates or input sensitive operators, a complete operator schedule can be statically constructed. This is done by calling the Schedule transformation, which applies a simple consistency predicate (time-stamp) to all the operators in the configuration.

The following three quotes, taken from papers written about DSEE and from DSEE manuals, give the reader an idea of how DSEE experts view scheduling.

> When you subsequently initiate a build, the configuration manager constructs a **partial ordering** of the components that need to be built. This partial ordering specifies which builds can be performed in parallel and which builds can't be started until other builds are completed[39].

> The alternative to concurrent programming, and the method used by DSEE CM [Configuration Manager], is to have a **scheduler** that knows which node to build (if any) at any point in the system build. The parallel builder calls the scheduler to determine if there is a node to be built and can handle the invocation and tracking of the build processes. It must also update the scheduler with the completion status of each finished build, so that the scheduler knows whether or not to schedule the component's ancestors[49].

> The DSEE parallel build scheduler quickly determines which component to build next by using a flattened, optimized scheduling structure. This structure uses ready lists and back pointers from each sub-component to its parent components (enabling quick updates to the ready list). The scheduling structure is created by walking the dependency tree **once**, before the first build is invoked[49].

Once the operator schedule is constructed, the Build transformation is applied.

To exploit DSEE's parallel builder, the assumption is made that the machine-count argument is greater than 1. With this assumption, the code that is executed in the Build transformation corresponds to Case 2 of Section 5.4.2 (no complex-predicates, no input-sensitive operators, multiple machines). Based on machine and operator availability, Build calls the OpSched object to apply the operators. As Lebang points out, the

> use of a separate scheduler makes the parallel builder architecture very straightforward. The parallel builder invokes builds as long as there are components ready to build and computers available to build them. After each invocation, the parallel builder processes any previously invoked builds that have completed. Completion processing includes updating the scheduler (to allow scheduling of builds that depend on those that have completed) and calling the resource selector to release the allocated compute slot.
>
> At any point when there is either nothing to build or no computer available to perform the build, the parallel builder waits for one of the builds already in progress to complete[48].

What actually happens for each operator application is similar to the explanation given in Section 5.4.1, as described in one of the DSEE manuals.

> Once the configuration manager determines that it needs to build a component it performs three steps.
>
> ...First it forms an **actual translation rule**. This involves substituting into the translation rule template .... The actual translation rule is a script that builds the component.
>
> ...Second, the configuration manager creates a new process and establishes a process context in which the desired element versions are automatically retrieved whenever elements are read by programs in that process.
>
> ...Third, the configuration manager invokes the actual translation rules in the specially prepared process environment[39].

Discussion. Because DSEE is built on top of an operating system that has expanded capabilities, and DSEE itself is a more integrated SCM solution, it is able to provide richer SCM capabilities than Make. The fact that the operating system (object repository) provides a single additional attribute for objects allows DSEE to support a restricted concept of generic components, and thus component families. This, plus the fact that version management is an integral part of the operating system, makes the concept of generic component models and their refinement to specific component models possible. Also, because DSEE provides derived object management (in terms of attributed derived objects) it is possible to conveniently compose and maintain multiple instances of a particular system that differ only in the parameters used to construct them.

Overall, DSEE is far superior to Make in terms of its expressive power. However, as one might expect, there are several significant problems with DSEE:

(1) It does not support user-defined attributes for components.

(2) Its concept of operators and operator inferencing is too restrictive.

(3) It does not provide effective support for software projects in which the number of versions of a system is large and the structure of individual versions of the system is highly varied.

(4) Because of its reliance on the specifics of the Domain/OS operating system, neither its specifications or its capabilities can be ported to another system.

As shown in the next two sections, the Adele system addresses issues (1) and (2), and issue (3) is extensively addressed by the Odin system.

6.3 Adele

Adele [23,22,40] is a software configuration management system developed at the Institute I.M.A.G. in France, to support the development and maintenance of modular programs. Adele's most powerful features are its attributed object repository, its concept of component families, and its ability to dynamically construct component models. The goals of the Adele project are to:

- Provide a database for the long term storage of components;
- Provide a language for describing system composition;
- Automate operations such as modifying versions of components and propagating their effects[40].

Adele can be viewed as taking the concepts of component model composition described for DSEE and generalizing them in two directions. Adele first generalizes the concept of component families. This is done by introducing interfaces and realizations as semantic entities and providing constructs for associating arbitrary user-definable attributes with components.

Secondly, Adele generalizes the way in which component models come about. Recall that in DSEE the SM contains a generic, statically defined component model (static gCM). The Adele group argues that when many versions/variants of systems exist, a static, generic definition of a system structure does not adequately reflect the complexity inherent in the real-world. To combat this complexity, and to make systems descriptions extremely short, they decided that the component model should be dynamically constructed based on the version/variant desired. In Adele, interfaces represent generic components, and their realizations (or implementations, or bodies) represent their concrete instances. As particular generic components are refined into specific realizations (the selection being based on constraints to which the software system must conform), the interfaces they in turn use/require are added to the model. This concept of constraint-based dynamic model construction is further augmented with the addition of object-specific constraints. Each specific realization has the capacity of imposing additional constraints on the rest of the composition process. Unlike DSEE, this implies that the initial constraint set may be augmented.

Because of its more generalized concept of component families, Adele provides a more generalized constraint language for refining them to concrete components. Details of the kinds of constraints supported are discussed below.

6.3.1 Semantic Entities

Virtually all of the information and power of the Adele system stems from the underlying object repository and the constraint-based system composition language. The following is a summary of the Adele semantic entities, which are discussed in detail in the paragraphs below:

- **Types:** Families, manuals, interfaces, realizations and Composite component constructors.
- **Relations:** Dynamic Composed-of.
- **Operator Graph:** None.
- **Constraints** Object-specific (c-c).

Types. Adele supports user definable composite types. It also supports the general concepts of interfaces and realizations (also called bodies or implementations)[23]. These concepts are similar to those found in more modern languages such as Ada, Oberon and Modula-2. Adele defines "manuals" as entities associated with objects that specify the **requires** information (used for selection-specific models) and object-specific constraints. The **requires** information corresponds to the Depends attribute, and the object-specific constraints correspond to the Constraints attribute defined for the GT Composite class in Section 4.2.1.

Adele also supports the concept of a "configuration" (not to be confused with the GT concept of a configuration). A configuration in Adele is similar to a Request Graph in the GT model. Initially it contains the root interface of the component model and the set of initial constraints. Because of Adele's limited operator capabilities, the product (a compiled and linked object) is implicit in the request. When satisfied, a configuration more closely resembles a specific component model (Section 2.2.1.1). When one issues an "exec" command against a configuration, the complete component model is constructed.

Relations. In Adele, component composition relations are dynamically constructed. The information used to construct these relations is found in the "manual" of selected objects. Details of this process are described below under composition.

Operator Graph. The Adele system is primarily concerned with the composition process. Based on the information available in the literature (including an Adele manual[21]) it appears that Adele has a very limited concept of operators. Within the literature there are no references to how operators or tools are specified. Operators appear either to be hard-wired compilers and linkers, or to exist outside the Adele database and are applied using the "exec" command[22]. The "exec" command allows arbitrary commands to be applied to a composition list.

Constraints. Constraints in Adele are partitioned into three sub-classes: imperative, exclusive and conditional[40]. Imperative and exclusive constraints correspond to the Depends attribute used to build selection-specific models discussed in Section 3.3.2. Conditional constraints correspond to the concept of object-specific constraints in the GT model. Estublier states that the constraints on a program component express restrictions on the objects that this component may (transitively) require. These constraints take the form of logical expressions involving conditions on attribute values. Thus one may expresses implications (e.g. version X of module A requires, or excludes, version Y of module B). Default conditions are also attached to each object; they are in effect if no other constraint is specified[40].

6.3.2 The Process

The Adele system's most salient feature is its dynamic composition process. The Adele authors have taken the view that generic configurations are too awkward to manage when large numbers of versions and variants exist. They argue that a specific component model should be dependent on the set of constraints to which the system must conform[22]. One of the principle goals of Adele was to design a composition language and supporting data model that allowed short, constraint-based descriptions of entire systems—a goal they achieved: Adele was used to manage a system comprised of more than one million lines of code, but the longest system specification was only nine lines long.

The following summarizes the processing capabilities supported in Adele. Their impact on the process is described below.

(1) **Object-specific Constraints:** Component-component.
(2) **Selection-Specific Models:** Supported.
(3) **Input-Sensitive Operators:** Not Supported.
(4) **Complex Predicates:** Not Supported.

The Process Program

Composition. In Adele, software systems are composed from very short system descriptions called configurations. Typically a configuration contains a single interface that represents the root component of a software system and a set of initial constraints to which the resultant software system must conform. Adele's definition for configuration is not the same as the GT definition—it corresponds more closely to the GT concept of a Request Graph in which the configuration is constrained to only contain the component model. For clarity, the term component model is substituted for Adele's configuration definition.

Starting with the root interface, Adele looks for a suitable realization within a family of realizations that satisfies the constraints. If it finds a satisfactory realization, it inspects the manual of that realization for information pertaining to selection-specific models and for object-specific constraints. The selection-specific model information associated with a realization describes the set of interfaces the realization requires. This information is used to build the component model. The object-specific constraints supported by Adele are restricted to the component-component sub-class. This is because Adele does not have effective operator support, and therefore the other two sub-classes, component-operator and operator-operator are meaningless.

```
FE-realization-1
 Attributes:
    machine=Vax;
    language=Pascal;
 Manual:
    requires:
       lexer-interface;
       parser-interface;
       special-interface;
    constraints:
       version=V4r3;
end;
```

```
FE-realization-2
   Attributes:
      machine=Vax;
      language=Fortran;
   Manual:
      requires:
         lexer-interface;
         parser-interface;
  constraints:
end;
```

Figure 50: An example of two Adele realizations

For example, assume that a Pascal-Vax-FE is desired. The Adele configuration would initially contain a generic FE-interface and the initial constraints "machine=Vax" and "language=pascal". The initial constraints are used to find a suitable realization of the FE-interface. Figure 50 depicts two (simplified) concrete realizations of the FE-interface. The FE-realization-1 object on the left would be selected because it satisfies the machine and language constraints defined in the configuration. Once the FE-realization-1 object is selected, its "requires" property is inspected. Note that this particular realization requires the inclusion of three other interfaces: lexer-interface, parser-interface, and special-interface. These three interfaces (and their relation to this realization) are added to the component model. Notice that the set of interfaces required by this realization differs from the set required by the other concrete realization. Before any of the other interfaces are refined into realizations, the constraints property of the FE-realization-1 object is also inspected. Notice the "version=V4r3" constraint specified in the last line of code in Figure 50. This constraint is added to the constraint set with the effect that for all subsequent refinements to succeed, the selected realizations must have a version attribute, and the value of that attribute must be "V4r3". The composition process terminates when all interfaces have been refined into realizations and no new interfaces can be added.

In Adele, the addition of a constraint cannot affect prior refinements. Further, Adele does not check the constraint set for satisfiability. So, if "P" is introduced as a constraint, and in a subsequent refinement "not P" is introduced, Adele has no knowledge that the constraint set is unsatisfiable. This is an issue, due to its computational complexity, that the designers of Adele have decided not to address.

Figure 51 depicts the GT process program for the Adele system. The process starts out by creating and initializing an instance of an RG. The root of the configuration is set to the Config-Root argument, which represents the root interface of the desired

```
Adele(Config-Root,Config-Constraints)  {
   RequestGraph.New(RG);
   RG.Config.SetRoot(Config-Root);
   RG.CS.Constraints.Add(Config-Constraints);
   Composed-of.New(REL);
   select-dynamic(RG,"component",REL, RG.Config.Root,
                  REL.Closure(RG.Config.Root));
}
select-dynamic(RG,Model,REL,Node,Iterator) {
   Refine(RG,Model,REL,Node);
   if ( Compose(RG,Model,REL,Iterator) = 0)
      select-dynamic(RG,Model,REL,
                     RG.Config.BFS("component",Node),
                     Iterator);
}
```

Figure 51: The Adele-GT Process Program

software system. The initial constraints are set to the Config-Constraints argument. The process program then creates an empty instance of the Composed-of relation called REL. The tuples in REL are dynamically constructed and represent the edges of the component model. At the heart of the Adele process is the dynamic construction of the component model. The component model is constructed by the select-dynamic sub-goal. Recall that select-dynamic was defined and explained in Section 5.3.3.

To construct the component model the Refine transformation is called to refine the root component. Within Refine, the code for both CASE 1 and CASE 2 (page 68) is applicable because Adele supports both object-specific constraints and selection-specific models. If the refinement is successful, then the Constraints attribute of the refined object is inspected. If any constraints exist, they are added to the RG constraint set. The Depends attribute of the object is then inspected. In Adele, all inner nodes in the component graph (tree) will have non-empty values for their Depends attributes (by definition, they require resources lower in the tree). The generic objects (interfaces) contained within the Depends attribute are used to dynamically construct the REL relation. After the Refine transformation terminates, the Compose transformation is called. In Compose, the newly added edges in the REL relation are used to augment the component model in the configuration. Select-dynamic is then recursively called. Note that the next node to be refined is determined by visiting the next node in the component model in the breadth-first traversal (`RG.Config.BFS("component",Node)`).

The following extended excerpt from Estublier is offered to substantiate that this is, in fact, the process which Adele supports.

The following algorithm constructs a configuration which implements a given interface, starting from a configuration specification. The system attempts to construct the transitive closure of the dependency relation by a breadth-first search. For each interface, it must select a single implementation. This is done as follows:

- 1) Process the consistency constraints that apply to the interface ... This processing is done in three steps.
  - (process conditional constraints) ... .
  - (process imperative constraints) ...
  - (process exclusive constraints) ...
- 2) Scan the component list constructed in step 1.
  - If a conflict was detected, the configuration is marked "inconsistent"; a warning is issued to the user.
  - If the list contains a single component, select it; if several choices are possible, use the default rules (e.g. most recent revision or user defined default rule).
  - If the list is empty, the configuration is marked "incomplete"; a warning is issued to the user[40].

Manufacture. When surveying the literature, one finds little mention of the manufacturing process carried out by Adele. Because Adele was originally designed to support a single language environment, compiling and binding were probably the only operators needed. Estublier in discussing reconstruction states that the

> ...algorithm proceeds in two steps. In the first step, the components which are involved in a modification (e.g. because they depend on a modified object) are marked. In the second step, the marked components are processed in bottom-up order (with respect to the dependency graph). Typical reconstruction algorithms (e.g. (Feldman 79)) use timestamps to detect modified objects.
>
> We have chosen to introduce a more elaborate status information, and to maintain this status up to date, including the automatic propagation of changes; however, the reconstruction itself, which is a fairly expensive operation, is only carried out on user request[40].

In another paper Estublier describes the 'exec' command. The exec command is used to apply arbitrary commands to component lists. "The Adele environment provides the "exec" command which, given the name of a configuration and a set of actions, will perform the actions on each component of the configuration[22]."

As it turns out, Adele's reconstruction algorithm calls a Make generator tool[24]. The input to the generator tool is the composition list and dependency information, extracted from the configuration. The tool then generates a Makefile, and calls Make to carry out the manufacturing process.

Discussion. The original Adele pioneered the concept of dynamically constructed component models, component families and the use of object-specific constraints to specify the desired product. This was possible because the Adele group developed their own attributed object repository.

Adele has evolved over the last few years. Adele II[6] refines several of the notions found in Adele I, as well as introducing new ones. The two most salient features of Adele II are programmable triggered relations[6], as in APPLA/A [76], and opportunistic processing using pre- and post-conditions, as found in Marvel [42].

One significant problem with Adele is its lack of sophisticated operator management. It is clear that configuration management systems must be able to support diverse tools. The next system, Odin addresses this issue.

6.4 Odin

The Odin system[13,14] specializes in managing complex manufacturing processes. Odin is by far the most sophisticated tool management system available today. An example of a complex manufacturing process managed by Odin is Eli[88,33]. In Eli, compilers are constructed from high level specifications. The Eli system defines approximately 70 tools (or tool fragments) and approximately 450 derived types (products and intermediate derived objects). Odin is used to process requests for products and manage all of the derived objects.

The Odin system consists of a specification language, an object base and a query interpreter. The specification language provides powerful constructs for describing derived objects and the tools needed to produce them. The object base supports and manages derived object families. The query interpreter accepts, and efficiently satisfies, requests for products (derived objects). A request, described functionally, contains a root component, the type of the desired product and (possibly) parameters. For example, the following query requests that from the Compiler.spec object (component), the system produce an "executable" product using the "debug" parameter in its manufacture:

```
Compiler.spec +debug :  exe
```

All the information needed to infer the manufacturing process to derive a product is centrally contained and managed in an object called a Derivation Graph (DG). The DG is Odin's most salient feature. It allows for the separation of declarative information about objects from algorithmic information about the tools that manipulate those objects[14]. Unlike Make and DSEE, information about type transformations are not kept with each component model description (in each Makefile or each system model). Instead, all transformations for all possible types and their interconnections are centrally described in the DG. When a request is made, the DG is consulted to infer the transformations needed to construct the product. Perhaps the best description of Odin comes from Clemm: "Odin can be thought of as an interpreter for a high-level command language whose operands are the various software objects in the data repository and whose operators are tool fragments[13]."

6.4.1 Semantic Entities

Whereas Adele provides complex object support for primitive components (user definable attributes, families, etc.), Odin focuses on the specification and manipulation of derived components. The Odin specification language[12] describes the semantic entities of interest. It contains simple specifications for primitive types, and productions for the specifications of derived types. Associated with a derived type is the tool or tool fragment used to construct it, and any possible parameters that may modify the behavior of that tool, and thus, the object produced by the tool. These parameters define the derived object families that Odin supports. When an Odin specification is compiled, a DG is produced.

The following is a summary of the Odin semantic entities which are described in detail in the paragraphs below.

- **Types:** Simple, composite and compound derived type constructors.
- **Relations:** Static Composed-of and Derives.
- **Operator Graph:** Supports type inferencing, equivalence and super-type relations.
- **Constraints:** None.

Types. In Odin, primitive components are called "atomic objects" and derived components are called "derived objects". The types of derived objects that can be defined include simple-derived, composite-derived and compound-derived. Simple-derived types correspond to the simple-operator class defined in Section 4.2.1.2. They have one input and one output. Composite-derived types are used to construct output objects that tuple-structured. For example, the output of a compiler might be a composite-derived object that includes a binary object, a listing object, and an error report object. Compound-derived types correspond to input-sensitive operators—the number and kind of objects produced cannot be statically determined. Clemm points out that

> Odin's object-modeling capabilities also support the specification of object aggregates—such as arrays and structures. One result is that Odin can manage the integration of tools, such as source text splitters, which produce an unpredictable number of objects as outputs. Earlier systems such as Make are unable to model such tools[13].

The only kind of family Odin supports is the one for derived types. There is no support for primitive component or operator families. The definition of a derived type contains the tool used to produce it. Parameters that modify the tool's behavior can be defined in the derived type specification. These parameters define the derived object families that Odin manages. Odin's mechanism for managing requests involving parameters is considerably more sophisticated than in DSEE. Odin provides a mechanism whereby parameters specified in a request are automatically propagated to all the operators that use them.

Relations. Odin has a distinct concept of a derives relation. The derives relation is contained in the DG. It is statically constructed when the Odin specification is compiled. This relation is used extensively in request satisfaction to infer the intermediate derived objects and operators needed to satisfy the request. The details of how the derives relation is used to produce a operator model is discussed in the composition section below.

Although capable of supporting manufacturing steps that extract parts of the structure of a component model ("include" dependencies), Odin does not fully support the concept of dynamic component model construction as described in Section 3.3.2. For the most part, the concept of a component model in Odin is embedded within the contents of primitive objects. For example, for the compiler described in Chapter 2, the component model would be detailed as a list of names (composition list) contained in a primitive object. Assume that "Compiler.specs" was the root component used to describe a compiler. In Odin, the contents of "Compiler.specs" would enumerate all of the specification objects used to construct the compiler. The value of "Compiler.specs" might look similar to:

```
Compiler.specs:
     Lexer-spec
     Parser-spec
     Seman-spec
     Opt-spec
     Code-Gen-spec
     Assembler-spec
```

Given just the root of a component model, there is no convenient way in Odin to dynamically infer the entire component model. Moreover, because the concept of primitive component families is not integrated into Odin, there is certainly no way to perform component model composition as in Adele, or for that matter, even the component composition capabilities that DSEE supports.

Operator Graph. Odin supports a complex operator graph called a Derivation Graph (DG). The DG supports complex components, complex operators, operator parameters, and all three OG relations: derives, super-type and equivalence. The Operator Graph defined in Section 4.2.5 is patterned after Odin's DG. Clemm describes the DG as the

> key construct ... which models the way in which tools can be synthesized. The edges in this graph correspond to types of objects. User requests are treated as a request for named objects, which are considered to be instances of the types contained in the Derivation Graph. Graph traversal is used to determine which tools must be used to create which specific instances of which intermediate types in order to eventually satisfy the dataflow needs of the tools that produce the requested objects.

One distinction between the Odin's DG and the GT model is that operators in an OG are first class citizens, whereas in Odin they are not. Because tool-type equivalence is supported (i.e. tool versions are not supported) in Odin's DG, every derived type definition also serves as an operator definition and as a consequence, it is not possible to support operator families.

The concept of operator families is just as important and useful as the concept of primitive component families and derived component families. One use for operator families is to support tool versions. Once versions of tools are supported, all of the problems of composition that exist for components (object-specific constraints, selection-specific models) may also apply to operator composition. If operators are not explicitly represented as first class objects, it is impossible to create instances of them and/or to associate them with specific constraints about what other tools or versions of tools they are (in)compatible with.

Constraints. Odin has no support for object specific constraints of any kind.

6.4.2 The Process

The SCM process that Odin supports is weighted towards manufacturing. In the section above, the properties of Odin's specification language were discussed. In this section, the process that uses this information is detailed. Of particular interest and importance is the inferencing mechanism used to construct the generic operator model. In Odin, this inference mechanism is encapsulated in the DG. In the GT model, it is encapsulated in the FindPath method of the OG. For the purposes of the GT model, Odin's component model (typically embedded in the contents of its primitive components) is described as edges in a static Composed-of relation.

The features of the composition process that distinguish Odin from the other systems is its heavy use of the operator graph to construct generic operator models, and its support of input-sensitive operators. Odin lags behind Adele and DSEE in component composition because of its lack of support for primitive component families.

The manufacturing process in Odin is more sophisticated than in the other systems because it supports input-sensitive operators and uses a complex predicate called value-equivalence to short-circuit operator applications. However, the Odin manufacturing process is less sophisticated than DSEE's in that it does not support parallel builds.

The following summarizes the processing capabilities supported by Odin. Their impact on the process is described below.

(1) **Object-specific Constraints:** Not Supported.
(2) **Selection-Specific Models:** Not Supported.
(3) **Input-Sensitive Operators:** Supported.
(4) **Complex Predicates:** Value-equivalence.

6.4.2.1 The Process Program

Composition. Odin provides limited support of component composition. For the most part, component models consist of atomic objects that contain the names of other atomic objects, as described above. Odin does provide a mechanism for extracting some of the dynamic dependencies involved in a component model. For example, given a list (composition list) of "c" atomic objects, it is possible to extract the files which they, in turn, include (by executing a manufacturing step that searches though the text looking for "include statements"). These dependencies can then be used to determine product consistency. This can be accomplished only because input-sensitive operators are supported. Because Odin has no built-in support for primitive component families, there is no concept of generic component models, and no support for either object-specific constraints or selection-specific models.

Odin provides a rich environment for operator model composition. It does this by using the type of the initial component and the type of the desired product and

```
ODIN(Root,Product,Parameters,Composd-of,OG) :-
   RequestGraph.New(RG);
   RG.Config.SetRoot(Root);
   RG.Config.SetProduct(Product);
   RG.CS.Constraints.Add(Parameters);
   Compose(RG,"component",Composed-of,
          Composed-of.Closure(RG.Config.Root));
   Compose(RG,"operator",OG
           OG.FindPath(RG.Config.Root,RG.Config.Product));
   Refine(RG,"operator",nil,Root);
   Merge(RG,OG,RG.Config.Product,EXPAND);
   Schedule(RG,value-equivalence,1);
   Build(RG,RG.OpSched.ReadyList.First,1);
   }
```

Figure 52: The Odin-GT Process Program

consulting the DG to infer a derivation path (possibly a graph) between them. Once the generic operator model is constructed, the parameters (in GT terms, the initial constraints) are used to refine the gOM into an specific operator model. The parameters are used to modify the behavior of operators, and to instantiate/name elements in derived object families.

Because Odin supports input-sensitive operators, it is not always possible to produce complete configurations during the composition process. Configuration completion must be carried out during manufacture. This case is discussed in the manufacturing section below.

Figure 52 depicts the process program for Odin. The process starts out by creating and initializing an RG instance with the Product, Root and Parameters arguments. Once the RG is initialized, the Compose transformation is called to construct the component model.

Constructing the component model is trivial. The component model is constructed by calling the Compose transformation. It simply extracts the edges from the Composed-of relation and inserts them into the configuration. Because there are no attributes associated with primitive components, there is no need (and/or way) to refine the component model.

The Compose transformation is then called again to construct the operator model. The iterator method used by Compose for operator model composition is the FindPath method of the OG discussed in Section 4.2.5. Recall that if the OG is consistent, and a path exists between the Root type and the Product type, then FindPath will yield the complete stream of derives edges needed to construct the product. These edges are used to construct the gOM in the configuration. Once the gOM is constructed, the Refine transformation is called to propagate the parameters found in the constraint set to the appropriate operators, thus constructing the sOM. Because object-specific constraints and selection-specific models are not supported, the refinement process is simple.

Once the sCM and the sOM are constructed, the Merge transformation is called ("action=EXPAND") to complete the configuration. If input-sensitive operators are present, Merge will be unable to statically construct a complete configuration at this time and is postponed until more information becomes available.

Manufacture. Odin supports dynamic manufacturing. In a discussion about Odin, Clemm states that: "Odin can also adjust its object-management strategy dynamically depending on the outcome of tool executions[13]."

Because the value-equivalence complex operator is used, the Build and Schedule transformations must interact. Value-equivalence implies that if the output of an operator is byte-by-byte equivalent to a prior application of the same operator, then all subsequent operators that exclusively depend on this operator's output should be removed from the operator schedule. Because input-sensitive operators are supported, the manufacturing and composition processes must interact. As the results of input-sensitive operators become available, the build process must call the Merge transformation to fill in the missing parts of the configuration that depend on this new information.

The manufacturing process begins by calling the Schedule transformation to construct an initial operator schedule. Because Odin does not support parallel builds, the machine-count argument will always be 1. Because Odin uses a value-equivalence complex predicate, the Predicate argument is set to reflect this. As in all other systems, the initial operator graph is constructed by applying the simple-consistency-predicate to each operator in the configuration. If the predicate fails, the operator is added to the schedule. In Odin, it is likely that the initial schedule will be modified due to value-equivalence, as intermediate results are produced.

After the initial operator schedule is constructed, the Build transformation is called. The Build transformation is the work horse of the Odin system. Odin makes extensive use of the code in the Build transformation depicted on page 84, and described in Cases 3 and 4 of Section 5.4.2.

Because Odin supports value-equivalence, it is necessary after every operator application to determine whether or not the results of the operator application differ from the previous time (if any) the same operator was applied. If the results are identical, then the potential exists that operators can be removed from the schedule. Recall that the code for Case 4 (denoted by the `CASE 4` comment in the Build transformation on page 84) applies the operator and waits for the result. When the operator application terminates, Build calls the ReSchedule method of the OpSched object to potentially recompute a new schedule based on the results of this operator application. If in fact the output of the operator application is equal to that of a previous application of the same operator, the operator schedule may be pruned.

A trivial example of the utility of value-equivalence can be demonstrated by a comment being added to a source object, and a product dependent on this source being requested. If the results of applying, say, a compiler operator to this source are identical to the last time the source object was compiled, then all subsequent operators that were placed in the schedule exclusively because the source file had been modified are removed. In fact, if this source object was the only object changed since the last request for the same product, all remaining operators would be removed from the schedule.

The value-equivalence plays a much more important role when composite output types exist. When a source, or sources, of an operator are changed, the application of the operator may only affect one of the many of the derived objects the operator produces. In the initial schedule, the assumption must be made that all derived objects produced by an operator application will be changed. As a consequence, all subsequent operators that depend on any of the outputs of this operator must also be added to the initial schedule. Only during the actual build, after value-equivalence is applied, can the determination be made as to which output objects were not affected and which subsequent operators can be removed.

The other case that Odin supports (Case 3) is one in which input-sensitive operators exist in the configuration. When an input-sensitive operator is encountered during the Build, the code denoted as `CASE 3` in the Build transformation would be executed due to the value of the delay attribute being TRUE. The Build transformation applies the input-sensitive operator and waits for the operator application to complete. It then inspects the operator schedule to determine the subsequent operators that depend on the output of this operator (it performs a transitive closure of the derives relation encapsulated in the Ops attribute of the OpSched object), and performs a Merge transformation to fill in the missing parts of the configuration that can now be filled in. Of course, in turn, Merge calls Compose and Refine to construct an sOM that is then merged into the original configuration. Because the Merge transformation has added new operators to the configuration, the operator schedule must be updated or re-scheduled to reflect this. Once the schedule has been updated, the next operator in the ready list (which has also been updated) is used in a recursive call to Build.

The Build transformation continues to walk all of the operators in the operator schedule, adding and subtracting operators based on newly acquired results, until the operator schedule is empty. At this point the product requested exists and the query terminates.

Discussion. Odin is a powerful system for describing and carrying out complex manufacturing processes. In terms of derived object support and type inferencing Odin goes well beyond anything else currently available. Moreover, its support for value-equivalence and input-sensitive operators adds considerably to the expressive power of the system.

As stated above, what Odin lacks is operator and component families, object specific-constraints and selection-specific models.

6.5 Summary

The primary purpose of this chapter is to validate the GT model. In the last four sections, four prominent SCM systems of varying complexity and functionality were described as GT model instances. Because the GT model provides a unified conceptual framework capable of capturing even the diversity and complexity found in these systems, detailed analysis of their differences and similarities is possible. The parameters of that analysis are the semantic entities they support, their process program of transformations and the exceptional processing capabilities supported by that process program.

Figure 53 summarizes the semantic entities supported by the various systems. As can be seen the table is quite sparse. (This is in part because the selected systems were chosen for their orthogonality.) Notice that none of the existing systems support user-definable relations. Also notice that no system other than the GT model supports either component-operator or operator-operator constraints. This is actually not surprising considering that no system supports the concept of operator families (denoted Op. under Families in the table).

In terms of operator graphs, Make and Odin are the only systems providing any type inferencing at all, and in the case of Make type inferencing is restricted to only a single step inference. Sophisticated operator graphs should be required in all future SCM systems.

Figure 54 summarizes the exceptional processing capabilities supported by the various SCM systems. Adele is the only system that supports object-specific constraints and selection-specific models, and Odin is the only system that supports complex predicates and input-sensitive operators. However, even in Odin the complex predicate is not under user control and therefore more restrictive than facilities in the GT model.

Schema Part 1							
	Types					**Relations**	
	User		**Families**			User	Dynamic
System	defined	Agg	Prim.	Derived	Op.	definable	construction
Make	–	–	–	–	–	–	–
Dsee	–	yes	–	yes	–	–	–
Adele	yes	–	yes	–	–	–	yes
Odin	yes	yes	–	yes	–	–	–
Model	yes	yes	yes	yes	yes	yes	yes

Schema Part 2						
	OG			**Constraints**		
System	Inference	Equiv	Super	c-c	c-o	o-o
Make	1-step	–	–	–	–	–
Dsee	–	yes	–	–	–	–
Adele	–	–	–	yes	–	–
Odin	complex	yes	yes	–	–	–
Model	complex	yes	yes	yes	yes	yes

Figure 53: Summary of SCM Semantic Entities

Processing Capabilities					
System	Complex Predicates	Input-sensitive	Object-Specific	Selection-Specific	Logic
Make	–	–	–	–	–
Dsee	–	–	–	–	–
Adele	–	–	yes	yes	mono
Odin	value equivalence	yes	–	–	–
Model	user definable	yes	yes	yes	either

Figure 54: Summary of SCM system processing capabilities

CHAPTER 7

RELATED WORK

This chapter reviews some of the related work not already discussed elsewhere in this thesis. The Chapter is decomposed into three sections. Section 7.1 details related work in the area of composition. Section 7.1 describes on-going research in the area of manufacturing. Finally, Section 7.3 summarizes this chapter.

7.1 Composition

This section describes research related to configuration composition, focusing primarily on primitive component composition. Operator composition is conspicuously missing. This is not surprising considering that, to the best of my knowledge, there is no literature available that defines and/or reasons about operator families, operator constraints or operator family selection rules.

The section is broken into two sub-sections. Section 7.1.1 describes systems that are either currently being researched or are available commercially. The general focus of these systems is to extend the capabilities available in either Make or DSEE. Section 7.1.2 discusses data models for version and/or variant management. With respect to the GT model, these models can be viewed as schemas or representations for primitive component families. The boundary between Section 7.1.1 and Section 7.1.2 is somewhat artificial, given that to a great extent, the expressive power of a data model (and the language used to select instances) defines the composition mechanism an SCM system supports.

7.1.1 Other SCM Systems

Several SCM systems other than Make, DSEE, Adele, and Odin are discussed in this section. For the most part, these systems are derivatives and/or hybrids of one or more of the above systems discussed in detail in Chapter 6. The systems discussed are: Shape/AFS, SMS, and Jason. All three systems support generic component models (gCMs) and a refinement mechanism (transformation) to select concrete component instances.

Shape/AFS extends Make by supporting user-definable attributes for components (component families) and is similar to DSEE in that it supports selection threads. SMS can also be thought of as a derivative of Make, DSEE and Odin. SMS augments Make by the inclusion of user-defined component families and dynamic source dependency extraction. SMS is a derivative of DSEE in that integrated support for version control is provided by the underlying operating system. Like Odin, SMS supports a centralized registry for tools. Jason, a proto-type system developed as part of Doug Wiebe's thesis is, in effect, a restricted form of an SCM system generator that provides facilities for describing generic templates (generic component models, similar to DSEE) and a full first order validation constraint language. Each of these system is detailed in the following sub-sections.

7.1.1.1 Shape/AFS

Shape/AFS[57,58], a derivative of both Make and DSEE, extends Make by providing an Attributed File Server (AFS) and a configuration management tool called Shape. The Shape/AFS environment supports primitive and derived component families, generic component models, and a simple constraint language for refining the gCMs into sCMs.

The basic process supported by the Shape tool is similar to DSEE (although syntactically closer to Make). It GT terms, it **Composes** a generic component model, then uses selection rules to **Refine** the gCM into a specific specific component model. Then, similar to Make, Shape labels (Merge(action= "LABEL") the graph with the appropriate tools and then propagates tool parameters. Unlike Make, the parameters are used to define derived object families.

Shape/AFS extends Make by supporting variant class definitions which define primitive component families. Because it supports integrated version and variant selection and management, the Shape tool facilitates the selection of instances within families. Shape's system description file, called shapefile, corresponds to an extended Makefile. A shapefile contains:

- A static component model;
- Selection rules;
- Transformation rules; and
- Variant definitions.

Selection rules have the basic form:

```
*.Suffix, attr(Name,Value), attr(Name,Value) ...
```

where "Suffix" defines the type, "Name" defines the name of the attribute and "Value" defines the value of the named attribute. Transformation rules, besides describing the tool to be used for suffix transformations, also define the parameters that the tools will recognize and Shape/AFS will record, thus defining derived object families. The variant section defines the set of possible attributes associated with a software systems. Variant classes are used to define primitive component families, like partitions in CMA and variants in the orthogonal version model,discussed in Section 7.1.2. Variant classes define mutually exclusive partitions of the component family space. Variant classes are used in the selection rules to prune the search space for a selection match.

7.1.1.2 SMS

SMS is a Software Management System that was developed by the BiiN corporation[71, 16]. Built on top of the BiiN/OS operating system (a capability-based object-oriented extension of Unix), SMS supports attribute-based version selection, dynamic dependency extraction and tool registration. SMS's configuration template, similar to a Makefile, supports the following semantic entities:

- **Configurations:** describe any other (sub-)configurations to be included;
- **Products:** specify the software products exported by this configuration;
- **Components:** enumerates the specifications for the (generic) elements of the component model. Each component specification contains: a name, a directory path (optional) and a version selection goal (optional);
- **Dependencies:** detail some or all of the "uses" and "includes" interconnections that exist between the components;
- **Build steps:** describe selection-specific operators similar to those found in Make; and
- **Tools:** specify the input, output and options of the tools specified in a build step.

The SMS design was guided by four general principles that simplify the representation of dependencies:

- Automatically compute dependency information when possible;
- Infer dependencies from other declarations when possible;
- Separate dependency declarations from rules defining how to build derived objects; and
- Express dependencies in a form natural to the application domain.

The composition process supported by SMS is novel. The components in a configuration represent the generic objects needed to produce the product(s). Dependency information such as what objects are "included" by certain versions of components may be defined statically within a configuration, or they may be extracted dynamically by language-specific pre-processors. The mechanism that SMS provides for interfacing pre-processors is a generalization of the transitive include-dependency extraction mechanism provided by the Odin kernel[14]. As components are pre-processed, their dependencies are entered directly into the dependency section of the configuration. This information is then used in the manufacturing process to aid in determining if a derived component is "out-of-date" with respect to its input dependencies.

The refinement of a generic component model to a specific one is performed by the **bind** command. Unlike Adele, Shape or DSEE, "version goals" are embedded in the component section of the configuration template; there is no separate language for specifying family selection. When the **bind** command is executed, the generic model is refined into a specific one by applying the version goals specified in the configuration for each component. If a different version of the configuration is desired (i.e., the user wants to apply a different "thread"), a different configuration must be constructed that contains the appropriate version goals for the same components.

Similar to Odin, tools may be registered with the system. Once registered, their use can be inferred during configuration construction.

Details of the operator graph supported by SMS are not clear from the literature. One can deduce that SMS supports multi-step inferencing, and it appears that the derived (tool)type constructors available are significantly limited, relative to those found in Odin[72].

The SMS manufacturing process, similar to Make, is augmented with "short-circuit" facilities. By selectively "blessing" a derived component or manufacturing step, it is possible to shunt operator applications. This facility is used to acknowledge that known inconsistencies exist[73] in the software and are to be ignored temporarily. The user, sure that these inconsistencies are irrelevant for some of the derived objects, selectively directs the system to bypass certain objects or interconnections during its computations. For example, the addition of a new declaration to a system-wide "header" file, although included by virtually every component, may only directly affect the one component currently being modified. Because SMS uses a hard-wired timestamp predicate to determine what operators to apply, if inconsistency management did not exist, all operators in the configuration that transitively depend on any of the components that transitively included the changed header file would have to be (re-)computed.

7.1.1.3 Jason

Jason[84] is an instance of Wiebe's model for generic software configuration management. It is perhaps the most complete and formal model available today for the less dynamic aspects of SCM[84]. It focuses on four basic concepts of SCM: classification, versioning, validation and build. The premise of the Wiebe's thesis is that the user/system administrator should decide what policies are appropriate for a particular site/project and that a language should exist that could be used to tailor an SCM system's needs to the chosen site. Some of the basic concepts of Wiebe's model and the Jason proto-type include:

- A theoretical model based on Universal Algebra and order sorted Σ-Algebras, that gives precise mathematical definitions for software objects, classes, object bases, versions, families and validation constraints;
- A language for specifying SCM schemas including component classes, static dependency relations and functions over class attributes;
- A formal model for generic component families as partial algebras, concrete instances as total algebras and algebraic closures to capture the refinement process;
- Static, generic system templates (describing system structure), as a partial closure and the refined template (configuration) as a total closure; and
- A model-theoretic approach, using constraints formulated in a full 1st order predicate calculus, to define configuration consistency.

Wiebe's primary contribution is an eloquent mathematical model for consistent composition. Its most salient feature is that it formally captures the intuitive notions of component families and their instances. Unfortunately, it may be extremely difficult to extended the model to include the more dynamic aspects of the SCM process in general, and in particular, to formally capture the mutually recursive dependencies between the various SCM sub-tasks as defined in the GT model.

The model assumes that static generic templates (or fixed partial closures) exist describing the generic component model. The template is then refined into a complete configuration (total closure). The configuration's validity (consistency) is then determined by using the configuration as an interpretation of the constraint formula[84]. These mathematical interpretations are possible because of two simplifying assumptions:

(1) The generic templates (gGMs) used, are statically defined, and selection-specific models are not supported. All of the generic components, the number and kind of their attributes, and the component interconnections can be statically determined and analyzed.

(2) The constraints used to validate the configuration are can be statically determined. Because object-specific constraints are not supported the constraint set used to validate a configuration is independent of the objects selected. As a consequence the constraints defined with the template, can be statically analyzed and compiled.

Essentially, Wiebe's thesis can be viewed as revolving around statically determined refinement and validation transformations over fixed generic component models. Refinement is similar to that found in the GT model, in DSEE and Shape/AFS. The most significant difference is that the selection threads exist as as part of the build plan (similar to Makefile or shape file or system model). This again is needed to have the information statically available to conform to the mathematical model. One the template is refined into a configuration it is validated. Given that the generic component model is fixed and all the constraints are statically known, it is possible to compile the validation constraints into C++ class objects. To validate, the configuration is passed as an argument to a generated C++ validation object, which yields TRUE or FALSE.

The actual build process leaves much to be desired[83]. In fact Wiebe states that the semantics of operator application (execution) are not defined in the formal model and that Jason does not actually perform any building. Using Jason, a tool called Jake was constructed (and customized) to model Make's build process. The actual build is performed as a by-product of the generated validation objects. This is done by a pairwise timestamp comparison between the each pair nodes in the configuration. Due to the way in which validation objects are implemented, it is possible that the same operator may be applied to the same source multiple times.

In order for Wiebe to be able to adhere to the formal rigors defined by his mathematical model, most of the dynamic policies needed in for complex real-world problems have been omitted.

7.1.2 Version/Data Models

The version/data models discussed below are the Orthogonal Version Model (OVM), the data model for the Configuration Manager Assistant (CMA) and the Change Oriented version Model (COM). These models are essentially refinements on the basic concept of component families. All support user defined attributes as found in Adele, Shape/AFS and SMS. Both OVM and CMA refine this concept by partitioning the set attributes of an object into equivalence classes, each of which has specific semantics. The COM presents a slightly different view.

7.1.2.1 Orthogonal Version Control

The Orthogonal Version control Model (OVM)[68], is yet another variant of attribute-based version management. The author, Reichenberger, argues that the tree structure of version and variants (called "intermixed organization") is insufficient to cope with today's software complexities. The "orthogonal organization" is similar to Wiebe's idea of partial instantiation and the concept of generic families as discussed in this thesis. Reichenberger states that the

> main difference between the orthogonal and the intermixed version organization is that the orthogonal organization, instead of managing versions of atomic objects only, also deals with variants and revisions of the whole software project. In fact, the management of versions of the whole software project is the basic idea of the orthogonal version management[68].

The object space in this model has three dimensions: **component**, **variant** and **revision**. In GT terms, a component corresponds to a component family and a variant corresponds to mutually exclusive sub-families, as in Shape/AFS and CMA. A revision is standardly defined as a delta made to component variants. Given these three dimensions, the following sets can be defined:

- $\mathcal{C}$: the set of all component-ids;
- $\mathcal{V}$ the set of all variant-ids;
- $\mathcal{R}$: the set of all revision-ids; and
- $\mathcal{O}$: the set of all object-ids

With these sets, a mapping function can be described that captures the essence of the refinement process[68]:

$$f\colon \mathcal{C} \times \mathcal{V} \times \mathcal{R} \rightarrow \mathcal{O}$$

7.1.2.2 The Configuration Management Assistant Data Model

CMA[66] is a configuration management system under development by Tartan labs. It can be viewed as an elaborate mechanism for name storage and retrieval [66]. Unfortunately the information provided about CMA is less detailed as compared to other systems due to its confidential nature as a Navy contract[65].

CMA attempts to provide a sufficiently rich data model to support various configuration management policies and processes. CMA focuses on the composition process. Within the composition process CMA is primarily concerned with component selection. It provides the concepts of logical objects and instances of those objects. Logical objects correspond to generic components in the GT model and instance objects correspond to concrete instances of generic component families. Ploedereder, one of the creators of CMA, states that

> CMA has the concepts of <u>logical objects</u> and their <u>instances</u>. A logical object may have multiple instances, distinguished by key attributes that characterize each instance. For example, different versions of an object will typically be instances of

the same logical object. .... The CMA model does not explicitly support a tree-structure of instances (i.e., versions of versions). Instead, multiple key attributes can be used to reflect subordinated versioning; nevertheless, all versions are regarded as direct instances of their logical object. This relieves the user of the necessity of an a-priori decision regarding such tree structures and avoids the potential need for entity duplication to conform to a tree-structured division of information[66].

One of the principle components of the CMA data model is the hierarchy (ordered partitioning) of attributes that can be associated with types. The hierarchy consists of:

- Partition;
- Rendition; and
- Version attributes.

Partition attributes are mutually exclusive. For example, if a system defines "machine" as a Partition attribute whose enumerated values are either "SUN" or "IBM", then a configuration in which "SUN" is specified as an initial constraint cannot include any component that has any other value for its "machine" attribute. This is similar to the concept of variant attributes as defined in the OVM described above. A Rendition attribute is similar in semantics to a type. There can be only one Rendition attribute (pre-)defined for objects. For example, DOC, SOURCE, TEXT are examples of Renditions. Finally there are Version attributes, which are used to distinguish instances in the same Partition and in the same Rendition.

Unique to the CMA data model is the notion of specialization and generalizations of attribute **values**, defining a partial ordering. For example, the "machine" attribute could have the two enumerated values: "stack-based" and "register-based". These values in turn can be specialized. For example, "register-based" could have specialized sub-values defined as "register-window", "32-registers" and "16-registers".

The CMA model also defines a set of relations that can be used to compose configurations. All of the CMA relations can be represented as binary relations whose domain and co-domain are either Logical or Instance objects. The relations provided are:

- **Component (Logical X Logical):** The component relation describes structural relations between families. The Component relation corresponds to the Composed-of relation in the GT model, used to contain the generic component model
- **Instance (Logical X Instance):** The instance relation describes membership in a family relationship. One inherent constraint in this relation is that instances can be a member of only one family.
- **Logical Dependency (Instance X Logical):** Logical dependencies correspond to the Depends attributes of Composite objects in the GT model, and are used for generating selection-specific models.
- **Inheritable Dependency (Logical X Logical):** The inheritable dependency relation describes relations that exists between logical objects. It states that whenever the first logical is needed the second one is needed as well and serves as

a template fro constructing logical dependencies. In GT terms, when an instance of the domain Logical object is created, the co-domain Logical object (actually a reference to) is copied into the Depends attribute of the newly created instance (thus forming a logical dependency)

- **Consistency**: (Instance X Instance) Consistency relationships only exist between instances of different logical objects or disjoint renditions. Consistency is used to capture the concept of (in)equivalence between instance objects between families and has no direct equivalent in the GT model.
- **Compatibility (Instance X Instance):** the Compatibility relation describes equivalence relations that may exist between instances within families. It states that the co-domain instance can be substituted by the domain instance in any configuration requesting the inclusion of the domain.

The CMA data modeling capabilities and the pre-defined relations provide extensive facilities for the composition process. They include mechanisms for describing generic families, describing semantics over the set of attributes that define families, representing generic Composed-of relations, object-specific constraints(in the restricted form of relations) and selection-specific models (also in the form of relations). How effective these constructs are will become apparent as CMA becomes available for use and data model is exercised by building real SCM systems.

7.1.2.3 Change Oriented Version Model

The Change Oriented version Model (COM)[51], which is part of the EPOS project[34,17,55,63], offers a mechanism for defining and accessing functional changes made to objects within a project base. Functional changes can be thought of as variants induced to one or more objects because of, say, a change in project requirements or the need to fix a bug. Similar to the mechanism found for the individual files used in MVPE[74][1], functional changes to an object are represented as option bits associated with that object. As in MVPE, all versions of one object are stored in one complex structure. By providing a predicate over these options, one or several versions of the file can be viewed/edited at the same time[2]. To some extent, functional changes correspond to partition attributes in the CMA model, in that they are global for an entire configuration. By selecting the UNIX option (or functional change attribute), the object space is partitioned to exclude objects that do not have this option.

7.2 Manufacture

The manufacture section is broken into two sub-sections. Section 7.2.1 discusses related work in the area of difference predicates and their relation to the GT model. Section 7.2.2 discusses complex manufacturing systems found in areas other than software configuration management and how they relate to the GT model.

[1]MVPE is a multi-versioned instance of IBM's personal editor that uses a single, sophisticated data structure to maintain all versions of source files simultaneously.

[2]This solves the problem of making an update to a base RCS or SCCS file and all subsequent versions. This is done by setting the predicate for the file to be edited to ALL,

7.2.1 Difference Predicates

The ability to support complex difference predicates (or simply complex predicates) in SCM systems is important if the goal is to minimize the amount of work needed to be performed. As stated earlier, the predicates value-equivalence[13], smart-recompilation[80], smarter-recompilation[73] and interface recompilation[62] are all instances of complex predicates. While value-equivalence applies to most manufacturing steps, the others typically involve single-language systems in which it is more convenient to acquire additional semantic information. In the research detailed below, Borison provides a general framework for describing complex predicates.

In her Model of Software Manufacture, Borison addresses the question: how much work is necessary to incorporate a given set of changes consistently into a given configuration by substituting one set of (derived) components for another[7]? To answer this question, precise definitions for configurations, manufacturing graphs and difference predicates are given. Difference predicates are used to compare related components and to discriminate between substantial and insubstantial changes.

Borison's configuration definition is similar to the one used in the GT model. Borison assumes that a configuration is complete, i.e., that it is bound and that all the information needed to carry out the manufacture of a product is contained therein. Borison's model does not address how such configurations come about, nor does it address the environment in which difference predicates can be applied. Borison defines a **configuration** as a tuple C = ⟨G,E,L⟩ where:

- A manufacturing graph G = ⟨C,M,I,O⟩, where G is a labeled finite bi-partite DAG with vertices C and M, and edges I and O, where:
  - C is the set of Components;
  - M is the set of manufacturing steps or operators;
  - $I \subset (CxM)$ represents the input relation; and
  - $O \subset (MxC)$ represents the output relation.
- $E \subset C$ are the exports or products.
- L is a labeling function for both components and operators.

Borison states that the

> structure of the manufacturing graph G captures the derivation relationships among the components of a system. If $D^* = (I \cup O)^*$ is the reflexive, transitive closure of the union of the input and output relations I and O, then a component c_i depends on a component c_j if $(c_i,c_j) \in D^*$. Similarly, a manufacturing step m_i depends on a step m_j if $(m_i,m_j) \in D^*$, etc. Since the graph is acyclic, D^* is a partial order[7].

A Difference predicate is defined to be an assertion about the relationship between two sets of components, and has the form:

- **P_{name}: (KxK) $\rightarrow$ {true,false}.**

Borison goes on to classify a hierarchy of predicates based on their strength[8]:

- Big Bang P:[true]
- Make P:[timestamp]

- Gratuitous (Headers) P:[white space,comments]
- Gratuitous (All files) P:[white space,comments]
- Name-use P:[smart-recomp(Headers)]
- Demi-oracle P:[smart-recomp(All files)]

Each predicate increases in computational complexity and the ability to discriminate unnecessary compilations. Borison's model centers around Name-use—a smart-compilation predicate, which determines which files need to be recompiled based on what names have changed and whether those names are (re-)used.

7.2.2 Complex Manufacturing

While Odin, and to some extent SMS, are the only SCM systems in which relatively complex operator graphs are used, two other systems, one in the area of vlsi chip design and the other in the area of post-processing very large scientific data sets, use dataflow analysis over tool sets to determine what operators need to be applied. In both cases, as in Odin/Eli[88], complex tool suites with complex input/output behaviors are managed by the system.

7.2.2.1 Tacos

TACOS [38] is a system developed at AT&T for managing the complex tool suites associated with performing integrated circuit (IC) design. In TACOS, the IC design process can be viewed as a series of data transformations that are represented by Data-Flow Diagrams. The Data-Flow Diagrams are used to capture domain specific knowledge which is then used to automate the process. Chang, one of the principle developers of TACOS, states that the

> ... IC Design process can be viewed as a series of data transformations and may be represented by a Data-Flow Diagram (DFD). A node in the DFD corresponds to a task-object transforming incoming data to outgoing data [operators] and an arc joining the nodes corresponds to a data-object. .... Construction of the DFD involves binding a set of task-objects, instantiation of relevant data/task-object, and linking them to appropriate task-objects. However, construction of a DFD for a complex design task from scratch is generally not a simple task for average designers. Designers intent on using CAD tools is [sic] usually expressed in vague, sketchy and incomplete terms. .... TACOS captures such an incomplete form of the intent [TACOS captures the user's incompletely specified goal] and generates a DFD based on a **goal-directed backward chaining inference** over the task-object library. Two task-objects can be chained if one's "in" slot matches with its predecessor's "out" slot. The chaining operation is repeatedly performed starting from "what they want to get" until it covers "what they have now" ....
>
> In most real-world design tasks, backtracking to a previous stage is inevitable when the result of the current stage is not satisfactory[38].

TACOS is a front-end interface to the IDEAS/Executive system also being developed for AT&T[25]. IC designers, like compiler designers (and users of the system above) deal with large numbers of off-the-shelf tools, versions of those tools, and complex tool interactions. The following excerpt could extended in scope to apply directly to software configuration management, instead of IC design:

> Bell Labs typifies the predicament facing a large number of industrial vendors who have been in the chip design for some time. They have large monolithic tools, which do not give in very easily to integration per se. Most tools create output files which are not compatible with the next tool in the design process, hence the proliferation of language translators. ...Translation steps can be hidden from the designer by encapsulating them within tasks. .... A common database eliminates redundant translations by allowing tools to deposit information directly into well defined data structures in the database file. Thus, consistency management may be enforced on the data flowing between tools. Explicit version control and a capability to track file versions during creation combined with a rich set of queries allows the user the capability to manage data from the script[38].

Complex manufacturing in SCM and in IC chip design appear to have much in common. Both take primitive components and apply complex sequences of tools to produce a product. Notice the correspondence between the DFD in the TACOS systems and an operator model in the GT model, also the TACOS task-object library and the GT operator graph.

7.2.2.2 The Tool Grid

At Lawrence Livermore labs, a system is being developed to automate the post-processing and formatting of very large scientific data sets to provide users with alternative ways of viewing their results [54]. In this environment, the users are not necessarily programmers and are not interested in knowing how a view is constructed or what intermediate representations are produced. When a user wants to see a particular view, a simple request for that view is made, and the system determines and carries out the manufacturing steps needed to produce the view. The tool grid[54] is the object that corresponds to the GT operator graph. Longstaff describes the tool grid as

> a collection of tools such that each output data set from a tool is connected to all the tools that can use this syntactic form as an input data set. This forms a directed graph with the tools as the nodes, and the data set types as arcs in the tool grid[54].

The manner in which the tool grid is used is also similar to a GT request in that the

> information required to select the appropriate intermediate tools [and determine the intermediate data sets] consists of the tool grid, the starting data set [root component], the desired display tool [the product] and a set of constraints on the data attributes [initial constraints] of the input data set of the selected display tool[54].

The algorithm used to find a path through the tool grid is described as follows:

> The tool grid is scanned to construct a list of all of the possible tool paths between this initial data set and the selected display tool. This construction of the tool paths is accomplished using the flow graph algorithm[3]. ....
>
> If more than one path between the initial data set and the display tool exists, constraints are placed on the attributes of the input data set of the display tool in an attempt to eliminate some of the tool paths that do not give the user the desired result. These constraints are a set of predicates to test the links between each pair of tools in a tool path and eliminate those that do not satisfy the constraints given by the user[54].

The tool grid presented by Longstaff, although somewhat specialized for scientific data manipulation, has many of the same basic characteristics as the operator graph of the GT model or the Derivation graph in the Odin System.

7.3 Summary

In this chapter several existing SCM systems, data models and models of SCM certain aspects of the SCM process were presented. Where possible references were made to their relevance to the GT model. The related work presented in this Chapter is either consistent with, or a refinement of the ideas and concepts presented in the GT model.

[3]The following is Longstaff's footnote: "This algorithm is used in algorithm theory to reduce a finite state machine (FSM) to a regular expression accepted by the FSM. It is fully described in[53]"

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

The industrial community is interested in trying to control the cost of, and to establish consistency in, the development and maintenance of large software systems. They are guided by practical concerns and the available technology. The SCM research community should take a broader view toward the future. A comprehensive model is needed that not only captures the salient features of current SCM systems/processes, but that can also used as a tool to reason about the capabilities of future systems. It should simultaneously capture the current and perceived future directions of SCM, so as to influence progress in directly related areas, such as programming languages, persistent data management, and operating system capabilities. Developing such a model need not be hindered by current "practical" concerns.

The GT model provides a starting point that captures the most salient features of existing SCM systems, and provides a design space capable of describing significantly more complex SCM systems. Future SCM systems, in which operators are raised to first class citizens, where operator families exist, and where more varied and powerful constraints are used to help manage (in)consistency, can be incorporated as natural extensions within the GT design space.

8.1 Conclusions

The primary contribution of this thesis is the Graph Transform model for the configuration management process. The thesis describes the GT model in terms of five mutually recursive graph transformations and the semantic entities they manipulate. The GT model, using object families, relations and an operator graph, transforms a user request into a configuration, generates an operator plan based on the configuration and then enacts that plan.

In its most elementary form, the GT model describes a basic generic SCM process in which a request for a software product is satisfied. Starting with the request, a generic component model and a generic operator model are composed, refined and then merged into a complete configuration. Once constructed, the complete configuration is then scheduled, built, and finally the product is made available. For the basic SCM process, the interactions between the various transformations are minimal.

In more sophisticated model instances, where object-specific constraints, selection-specific models, complex predicates and input-sensitive operators are present, the potential for complex recursive interactions between the transformations exists. Due to its unique organization, the GT model makes these interdependencies explicit in a clear and concise manner. Characterizing the nature of these innate interactions is important for anyone designing future SCM systems. While there is no system available today that provides all of the processing capabilities the GT model captures, future systems will.

The GT model provides a design space capable of describing more complex SCM systems than exist today. When compared against the full expressive power of the GT model, each SCM system modeled in Chapter 6 is deficient in certain processing capabilities. Moreover, the GT model is flexible enough to capture future SCM systems in which operators are raised to first class citizens, where operator families exist, where more varied and powerful constraints are used to help manage (in)consistency, and even systems where SCM policies can be changed by varying the interpretive process by which requests are satisfied.

Dynamic, interpretive SCM systems are needed in order for systems to shift as a software project(s) develops over time. At present, dynamic, interpretive SCM systems are perhaps too computationally expensive to be practical. Recall, though, that the same thing was said about Lisp systems only a few years ago.

The GT model develops the concept of operator families, operator-operator constraints and component-operator constraints. Operator families are used to define versions and variants of tools, just as primitive component families are used to define versions and variants of components. Once operator families exist, it is possible to analyze and manipulate them in ways similar to primitive component families. Operator-operator constraints, mirroring component-component constraints, can be used to describe complex relations between versions of tools. Component-operator constraints are introduced to model complex relations between component and operator selections. Using these concepts four existing systems can be captured and discussed as instances of the GT model. The fact that the GT model encompasses four disparate SCM systems reveals the power, breadth and depth of the model. This, combined with its future-oriented design space which allows for inevitable SCM complexities, makes the GT model an exemplary design objective for future implementations of SCM systems.

8.2 Future Work

The GT model satisfies the initial goals of the thesis: it describes potentially complex interactions between the various SCM tasks, it is able to model existing SCM systems and it suggests design points for future systems. However, given the nature of the GT model, there is ample room for future work.

The Validate Transformation

Between the composition and manufacture tasks, it is possible insert another transformation called Validate without affecting the structure or interactions of the other transformations. This transformation is similar to Wiebe's concept of validation[84]. The Validate transformation would use as input a complete configuration and a set of validation constraints. In order to integrate the Validate transformation into the GT model, the set of attributes defined for object classes would have to be partitioned into two sets: those used for family identification and those used for analysis. Identification attributes would be used in the Refine transformation, and analysis attributes would be used in the Validate transformation.

Because of the possibility of selection-specific models, constructing and compiling the appropriate validation constraints would be difficult, if not impossible. In this respect, the GT model greatly differs from Wiebe's model, where all the information needed to describe the construction of a complete configuration is found in one description object that can be statically analyzed. Because of the static nature of Wiebe's model, it is possible to use first order model theory (universal algebra plus first order logic) to guarantee the decidability of the constraints. When the templates and closures can not be statically determined, finiteness and decidability can no longer be statically guaranteed.

Selection-Specific Operators

In this thesis, selection-specific operator models are composed by refining generic operators into concrete operators and then using the value of the Depends attribute of the chosen operator to augment the shape of the model. Component-operator constraints were introduced so that the specific concrete components could affect the refinement of the operator model, but not the operator model's shape. What is not adequately covered is the case where a choice of a concrete component can also change the shape of the operator model. This was alluded to in Section 6.1 when discussing Make's explicit transformations. The GT model should be able to handle this case even though most of the need for such explicit transformations are eliminated with the introduction of set-operators and input-sensitive operators, and type inheritance.

One potential way to solve this problem could be to associate another attribute, the Depends-Operator attribute, with component objects. This attribute would describe the specific operator(s) needed by the specific concrete component, just like the Depends attribute of an component describes the other component(s) needed. The Refine transformation would have to be changed to inspect this attribute and perform the necessary computations:

```
  ∀ o ∈ obj.Depends-Operator
        RG.Config.InsertEdge("operator", MakeEdge(obj,o));
```

The InsertEdge method of the Configuration class would augment the operator model with the concrete operator(s) and the appropriate input relations.

If the entire operator model was constructed in this fashion, the impact would be minimal. On the other hand, what the impact would be if the operator model was partially specified by selected components, partially by the operator graph and partially by selected operators is unclear and is an area that should be investigated.

An Implementation

Finally, the most obvious area of future work is the construction of a working implementation of the GT model. However, before this can be done it is necessary to provide a sophisticated language/environment capable of encoding the complexity of the graph transformations in a clean, precise manner. Section 5.1 discussed the requirements of the language in which the graph transformations are written and the environment in which they execute. The language, characterized as a union of an object-oriented programming language (such as SmallTalk) and a logical or relational language (such as Prolog), is executed in a persistent, interpretive environment.

Object-oriented constructs provide mechanisms for encapsulation, locality of reference and a restricted form of inheritance[1]. Logical language constructs are needed to provide logical variables and backtracking, which are essential to anything but the most simplistic forms of constraint satisfaction. Finally, the environment needs to be interpretive in order to facilitate dynamic typing and the run-time ability to alter the SCM request satisfaction process.

Today, a language/environment providing a unified view of these features does not exist. One can argue that it would be possible to bring together elements of these paradigms to provide an language/environment capable of supporting the GT transformations. As Goguen points out, "the result would be a complex language with little or no intellectual coherence[31]". Therefore, significant research needs to be carried out to define the appropriate language/environment before actually proceeding to a GT implementation.

[1]Inheritance, as provided by object-oriented languages available today, is too restricted for the inheritance needs of the GT model. Current object-oriented languages are not adequate to express inheritance relations between successive refinements of partial families, or between sorts and their subsorts.

BIBLIOGRAPHY

[1] American National Standards Institute. *American National Standard for Informmation Systems - Database Language - SQL.* American National Standards Institute, Inc., New York, 1986. ANSI X3.135-1986.

[2] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA*, pages 420–440, October 1987.

[3] B. Asbrock, U. Kastens, and E. Zimmermann. *User Manual for the GAG - SYSTEM.* Technical Report 15/18, Universitat Karlsruhe, Postfach 6380, D 7500 Karlsruhe 1.

[4] M. Atkinson, F. Banchilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. of the DOOD Conference*, pages 40–57, Kyoto, Japan, December 1989.

[5] Eric H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, Spring 1988.

[6] N. Belkhatir and J. Estublier. Nomade: a kernal for programming in the large. draft.

[7] Ellen Borison. Software manufacture. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[8] Ellen Ariel Borison. *Program Changes and the Cost of Selective Recompilation.* PhD thesis, Computer Science Department, Carnegie Mellon University, July 1989. CMU-CS-89-205.

[9] Allen Borning. *Thinglab—A Contraint-Oriented Simulation Labratory.* PhD thesis, Stanford University, Palo Alto, California, 1979. Technical Report STAN-CS-79-746.

[10] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the exodus extensible dbms. In *Proc. of the International Workshop on Object Oriented Database Systems*, pages 52 – 65, 1986.

[11] Michael J. Carey, David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and file management in the exodus extensible database system. In *Proc. of the Twelfth International Conf. on Very Large Data Bases*, pages 91 – 100, 1986.

[12] Geoff Clemm. The odin specification language. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[13] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[14] G.M Clemm. *The Odin System - An Object Manager for Software Environments.* PhD thesis, Department of Computer Science, University of Colorado, Boulder, Colorado, 1986.

[15] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1981.

[16] Ellis Cohen, Dilip Soni, Raimund Gluecker, William Hasling, Robert Schwanke, and Michael Wagner. Version management in gypsy. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 201–215, November 1988.

[17] Reidar Conradi, Anund Lie, Espen Osjord, and Per Westby. *Software Process Modeling in EPOS.* Technical Report, Norwegian Institute of Technology, 1989.

[18] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, December 1987.

[19] Pamela Drew, Roger King, and Jonathan Bein. A la carte: an extensible frameowrk for the tailorable construction of heterogeneous object stores. In *Fourth International Workshop on Persistent Object Systems*, pages 233–246, September 1990.

[20] V. B. Erickson. Build - a software construction tool. *AT&T Bell Labs Technical Journal*, 63(6), July/August 1984.

[21] Jacky Estublier. *Adele II commands.* I.M.A.G.

[22] Jacky Estublier. Configuration management: the notion and the tools. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[23] Jacky Estublier. A configuration manager: the adele data base of programs. In *Workshop on Software Engineering Environments for programming-in-the-large*, pages 140–147, Harwichport, Mass, June 1985.

[24] Jacky Estublier. Personal communication. December 1. 1990.

[25] N Srinvas et.al. Ideas – an integrated design automation system. In *Proceedings of ICCD*, pages 398–402, October 1987.

[26] EXODUS Project. The E reference manual. EXODUS Document, May 1989.

[27] Stuart I. Feldman. Evolution of make. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[28] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–265, April 1979.

[29] G. Fowler. The fourth generation make. In *Proceedings of the Summer USENIX Conference*, Summer 1985.

[30] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, 1978.

[31] Joseph Goguen and Jose Meseguer. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*, pages 417–478. MIT Press, 1987.

[32] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[33] Robert W. Gray, Vincent P. Heuring and
Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, unknown, unknown 1991. [To be published].

[34] Svein O. Hallsteinsen and Ole Solberg. *Configuration Management - Concepts and Practice.* Technical Report, University of Trondheim, Trondheim, Norway, 1988.

[35] Karen E. Huff and Victor R. Lesser. *Meta-Plans that Dynamically Transform Plans.* COINS Technical Report 87-10, University of Massachusetts, Amherst, MA, November 1987.

[36] Richard Hull and Dean Jacobs. *Toward a Formalism for Module interconnections and Version Selection*, pages 423–440. *Frontier Series*, ACM Press, 1990.

[37] Andrew Hume. Mk: a successor to make. In *Proceedings of the Summer USENIX Conference*, Phoenix, Arizona, June 1987.

[38] Hyun-Taek and Kwok Wu. *A Knowledge-Based Approach to Design Task Configuration*, pages 21–27. Elsevier Science Publisher B.V., New York, 1989.

[39] Apollo Computer Inc. *Engineering in the DSEE Environment.* 330 Billerica Road Chelmsford, MA 01824. Order Number 008790-A00.

[40] S. Krakowiak J. Estublier, S. Ghoul. Prelimianry experience with a configuration control system for modular programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984.

[41] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report.* Springer-Verlag, New York, second edition, 1974.

[42] Gail Kaiser and Peter Feiler. An architecture for intelligent assistance in software development. In *Proceedings 9th International Conference on Software Engineering*, pages 180–188, Monterey, California, March 1987.

[43] Alfons Kemper, Guido Moekotte, and Hans-Dirk Walter. *GOM: A Strongly Typed Persistent Object Model With PolyMorphism.* Technical Report, Universitat Karlsruhe, Karlsruhe, West Germany, 1989?

[44] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language. Software Series*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[45] Butler Lampson. The practical use of a polymorphic applicative language. In *Proceedings of 10th POPL conference*, January 1983.

[46] David Leblang. Computer aided software engineering in a distributed workstation environment. In *SIGPLAN notices*, May 1984.

[47] David Leblang. The domain software engineering environment for large scale software development. In *IEEE Conference on Workstations*, November 1985.

[48] David B Leblang, Robert P. Chase, and Howard Spike. Increasing productivity with a parallel configuration manager. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[49] D.B. Leblang and Chase R.P. Parallel software configuration management in a network environment. *IEEE Software*, 4(6):28–35, November 1987.

[50] C. Lecluse, P. Richard, and F. Velez. *O2, an Object-Oriented Data Model*, pages 227–236. *Data Management Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[51] Anund Lie, Reidar Conradi, Tor Didrksen, and Even-Ande Karlsson. Change oriented versioning in a software engineering database. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 56–65, 1989.

[52] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge MA, 1986.

[53] Thomas Longstaff. *Attribute Constraints in User Environments.* PhD thesis, University of California, Davis, December 1990.

[54] Thomas Longstaff, Meera Blattner, and Jack Milton. *Scientific Data Manipulation using Attribute Constraints.* Technical Report, Lawrence Livermore National Labratory, 1989.

[55] Chunnian Lui, Reidar Conradi, Espen Osjord, and Per Westby. *A Software Process Planner in EPOS.* Technical Report ???, Norwegian Institute of Technology, Trondheim, Norway, May 1990.

[56] J. Rochkind M. The source code control system. *IEEE Transactions on Software Engineering*, 1:364–370, 1975.

[57] Axel Mahler and Andreas Lampen. An integrated toolset for engineering software configurations. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston Massachusetts, November 1988.

[58] Axel Mahler and Andreas Lampen. Integrating configuration management into a gerneric environment. In *Proceedings of the 11th International Conference on Software Engineering*, May 1989.

[59] D. Maier, J. Stein, A. Otis, and A. Purdy. *Development of an Object Oriented DBMS.* Technical Report TR CS/E-86-005, Oregon Graduate Center, April 1986.

[60] D. McDermott and J. Doyle. Nonmonotonic logic. *Artifical Intellegence*, 13:41–72, 1980.

[61] R Moore. Sematical considerations on nonmonotonic logic. *Artificial Intelegence*, 25:75–94, 85.

[62] Hausi A. Muller and Karl Klashinsky. *Rigi A System for Programming-in-the-large.* Technical Report DCS-77-IR, University of Victoria, Victoria, B.C., Canada, V8W 2Y2, February 1988.

[63] Jurgen Muller. *Process Management Using AI Planning Techniques.* Technical Report EPOS-86, Norwegian Institute of Technology, Trondheim, Norway, June 1989.

[64] Mary Patric Pfreundschuh. *A Model for Building Modular Systems Based on Attribute Grammars.* PhD thesis, Department of Computer Science, University of Iowa, Iowa City, Iowa, 1986.

[65] Erhard Ploedereder. Personal communications. January 1990.

[66] Erhard Ploedereder and Adel Fergany. The data model of the configuration management assistent (cma). In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 5–14, 1989.

[67] Prieto-Diaz. *Module interconnection languages, A survey.* Technical Report, Department of Information and Computer Science, U.C. Irvine, CA 92717, August 1982.

[68] Christoph Reichberger. Orthogonal version management. In *Proceedings of the 2nd International Workshop on Software Configuration Management*, pages 137–140, 1989. (In [85]).

[69] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis of language based editors. *ACM Transactions on Programming Languages and Systems*, 1988.

[70] Craig Schaffert, Topher Cooper, and Bruce Bullis. An introduction to trellis/owl. In *OOPSLA*, pages 9–16, September 1986.

[71] R. Schwanke, E. Cohen, R. Gluecker, W. Hasling, D. Soni, and M. Wagner. Configuration management in biin sms. In ACM, editor, *11th International Conference on SOFTWARE ENGINEERING*, pages 383–393, ACM, May 1989.

[72] Robert Schwanke. Personal communications. November 1990.

[73] Robert Schwanke and Gail Kaiser. Technical correspondence: smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.

[74] John Shilling and Peter Sweeney. Maintaining versions in a structure-based environment. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[75] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[76] Stanley M. Sutton, Jr. *APPL/A: A Prototype Language for Software-Process Programming.* PhD thesis, University of Colorado, August 1990.

[77] Swammy. Version control in the common lisp framework. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[78] W. Tichy. Tools for software configuration management. In *Proceedings International Workshop on Software Version and Configuration Control*, Grassau, West Germany, January 1988.

[79] Walter Tichy. Design, implimentation and evaluation of a revision control system. In *Proceedings 6th International Conference on Software Engineering.*, Tokyo, Japan, September 1982.

[80] Walter Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[81] Walter Tichy and Wolfgang FaltenBacher (scribe). Summmary of plenary discussion: "what is a configuration". In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[82] William. M Waite and Gerhard Goos. *Compiler Construction.* Springer-Verlag, New York, 1984.

[83] Doug Wiebe. Personal communications. March 1990.

[84] Douglas Wiebe. *Generic Software Configuration Management: Theory and Design.* PhD thesis, Department of Computer Science, University of Washington, Seattle Washington, 98195, 1990. Tech. Report 90-07-03.

[85] Jurgen F. H. Winkler, editor. *Proceedings of the 2nd International Workshop on Software Configuration Management*, ACM Press, Princeton, New Jersey, October 1989. ACM SIGSOFT SOFTWARE ENGINEERING NOTES Volume 17, Number 7.

[86] Patrick Henry Winston. *Artificial Intelligence.* Adison Wesley, 1984.

[87] Niklaus Wirth. *MODULA-2*. Technical Report Report 36, Institut fur Informatik, ETH, Zurich, 1980.

[88] U. Kastens W.M. Waite, V.P. Heuring. Configuration control in compiler construction. In *Proceedings International Workshop on Software Version and Configuration Control*, Grassau, West Germany, January 1988.