MODELING THE CONFIGURATION MANAGEMENT PROCESS

Steven P. Krane

CU-CS-427-89

Modeling the Configuration Management Process

Steven P. Krane

CU-CS-427-89 March, 1989

Abstract

This paper proposes a model for the software configuration management process. Viewed as a transaction, the process begins with a high level specification for a product and terminates when the product is made available. The model describes this process as a series of well-defined transformations to a graph. The two principle phases of the model are configuration construction and configuration execution. The construction phase builds up a complete configuration from the high level specification. The execution phase describes how the product is then made available from that configuration. Finally, an outline for PhD research based upon this model is given. The principle objectives proposed are to build a prototype of the model and to explore optimization issues.

Contents

1 Introduction

The purpose of this research is to formulate and validate a model for software configuration management environments. Five high level goals guide the work:

- Capture relevant notions from a wide set of existing systems
- Describe a general architecture for future systems
- Support system evolution
- Provide insight into the optimization process
- Define and exploit a transaction orientation

A model of software configuration management should be able to capture and explain the relevant details of existing systems. Existing configuration management systems cover a diverse and seemingly disparate spectrum of concerns and functionality. In fact, the spectrum is so diverse that one could ask the question: What do they have in common? That question started this research and it is the question that must be kept in mind throughout the investigation. While it is not possible to explain every detail of every existing system, the model is expected to capture and unify key concepts found in them.

The model should be sufficiently powerful and flexible to explore and research future systems. The model should suggest a logical decomposition that can have many different physical implementations. For small-scale projects one could envision implementations in which several logical tasks are folded together. In very large systems it might make sense to make each logical task into a separate server.

As a software system evolves and shifts focus it is important that the configuration management environment maintaining that software be able to evolve as well. In the evolution of many software systems the system model, a description of how the pieces fit together, is a rapidly changing entity. In early stages of software development it may make sense to pay the price for dynamic construction. Later on, when the software is sufficiently stable, resources might be shifted more towards speeding up the build process.

The configuration management system should also be flexible enough to allow for varying degrees of analysis. At times it might make sense to devote resources to full semantic analysis. At other times, especially when the system is known to be inconsistent, simple analysis might be sufficient.

In order for configuration management systems of the future to cope with the demands imposed on them by larger and more complex software, efficient ways of constructing and executing user requests must be explored. Optimizations need to be developed for all stages in the construction and execution of a configuration request. One of the initial insights that led to this research was that a configuration graph seemed quite similar to the data flow graphs used in compiler optimization. This insight inspired the idea of creating a unifying framework in which optimization research could be conducted.

Software configuration management is inherently transaction oriented: The user makes a request to the system, the system processes the request and the request either succeeds or fails. If one views the problem from a database perspective the request can be regarded as a transaction. With this view in mind all of the issues of locking, journaling, commit, and abort as found in the database domain apply. Moreover, if the environment on which the transaction is taking place is distributed then all the relevant notions of distributed transactions also apply. Of course there are differences. Exactly what these differences are and how they perturb the database definitions need to be explored.

Defining a model of software configuration management is important. A model provides a conceptual understanding of the task is being modeled. A model is an abstraction mechanism; it focuses on important issues, allows for the separation of concerns, and clarifying potential interrelationships. The model of software configuration management proposed below is intended to be such an abstraction mechanism.

1.1 Software Configuration Management

Software configuration management (SCM) is the discipline of software engineering concerned with the efficient, automatic construction of software products. It is also responsible for the maintenance and management of the entire collection of software objects needed to produce those 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 objects. 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.

Software Product Manufacture, sometimes termed software manufacture, is the process of producing a software product. The process is largely automatic and may involve a complex sequence of manufacturing steps, large tool sets, and large numbers of software components. Software product manufacture establishes relationships between the primitive software components that make up software products, the manufacturing steps needed for construction, and the software products themselves[Bor86]. These relationships are important because they supply the foundation necessary for automating the manufacturing process.

The manufacture of software objects is complicated by several factors. Loosely speaking these factors can be decomposed into two classes: those that revolve around a static view of software product manufacture and those that stem from a temporal dimension superimposed over that static view. Perhaps the best way to understand this distinction is to visualize a software system with which you are familiar . Take a snapshot of that system in time. Within that snapshot there is a whole class of factors that complicate the manufacturing process: the size of the system, the number of components involved, the manufacturing steps needed, and the relationships among the components, manufacturing steps and products. These can all be viewed from within the context of software product manufacture. Now think about that same system over the course of its lifetime. When looking at the system from this perspective, one thing is certain: change. In any two snapshots of the system, any or all of the factors mentioned above may differ: the number of components, the contents of the components, the introduction of new manufacturing steps, and even the introduction of new products.

Supporting a static view of software product manufacture is a complex, time consuming process. Some of the factors that contribute to this complexity are the size of today's software systems, the complexity of the relationships between the parts that constitute these systems, the number of manufacturing steps needed to produce the desired product and the tools that are needed to support the manufacturing process.

When discussing large systems it is presumed that:

1. Thousands, if not tens of thousands, of primitive components exist that comprise a software product.

2. Relationships exist between these components that constrain the manufacturing process.

3. These relationships are so complex that for most large systems it is impossible for a product to be produced correctly without computer aid.

4. Manufacturing an entire system may take days or even weeks to complete.

5. Thousands of intermediate objects are produced in the process.

Change over time adds an entirely new dimension to the management of an already complex problem. Change is inevitable. Requirements change, systems evolve, bugs need to be fixed. Not only must a configuration manager be able to manufacture a current product, it must be able to go back to a previous state (time) and manufacture a product relative to that state. This implies that the configuration manager must track and manage temporal relations among all of the objects involved.

1.2 The Software Configuration Management Process

A software configuration management transaction begins with a user's request for some product and ends when that product is available. The request is a high level specification containing enough information to uniquely identify the desired product. Throughout the process a knowledge base is consulted to supply information needed to complete the request. This transaction can be viewed as a succession of tasks that are to be performed in order to create the product. First, the high level request must be transformed into a complete request. The complete request must then be optimized, scheduled and executed.

The first significant task in this process is to take the user's request and add whatever information is needed to complete the request. A user request must contain at least an indication of the product desired and one or more primitive objects [1] from which the product can be derived. This information, in conjunction with facts found in the knowledge base, is then used to infer:

1. The sequence of manufacturing steps needed to produce the desired product.
2. The complete set of primitive objects that will drive that manufacturing process.
3. All of the intermediate objects produced as the result of applying the manufacturing sequence to the set of primitive objects.

As an example, imagine a request in which the user supplies two C programs as the primitive objects from which an executable product is desired. The first step would be to determine whether a manufacturing sequence exists. Using the knowledge base we can infer that c-objects can be compiled into o-objects and that o-objects can be used to create executable-objects. The second step would be to determine the complete set of required primitive objects. Suppose that each of the c-objects depends on one h-object (header file). The result of this dependency would be to add the h-objects to the initial set of primitive objects. Finally, The third step would be to infer that two intermediate o-objects will be produced.

The request may also contain attributes. Attributes serve to constrain or augment the inference processes. So (for example) if the user wanted the executable product to contain

[1] With respect to the manufacturing process.

debug information, an attribute "debug" would be made part of the initial request. (The way in which attribute values are propagated is the subject for later discussion.)

Once it has been determined that the request is legal, and that sufficient information has been obtained to complete it, the optimization task takes over. The complete request describes everything necessary to construct the product. In many cases, based on the state of the knowledge base, much of this information may already be available. For example, some intermediate objects might exist in the knowledge base and need not be reconstructed. In this case, parts of the complete request can be removed.

After the request has been pruned of all unnecessary manufacturing steps it must be scheduled. Dependency information and the number and location of certain resources put constraints on the order in which manufacturing steps can be executed.

Finally, the complete, optimized, scheduled request is ready to be executed. The manufacturing steps are applied, intermediate objects are produced and added to the knowledge base, and the desired product is made available. At this point the transaction is complete and the process is ready to begin once again.

2 General Architecture

Before discussing the architecture of the model it seems necessary and appropriate to describe a few terms and concepts. Within the model, all specifications take the form of attributed graphs consisting of *nodes*, *edges*, and *attributes*. The nodes represent software objects, the edges represent relations between software objects and the attributes represent characteristics of software objects.

Software objects are unique, typed, attributed objects. Software objects include concrete objects such as a particular source file, as well as more abstract objects such as types and configurations. Software objects may be very complex[Tic88,Cle88a]. Some examples of possible software objects are: class or type definitions, source code, binary code, sets of regression tests, configuration descriptions, module interfaces and module bodies. Figure 1 represents a simple taxonomy of software objects. Three major software-object categories of particular importance in the sections below are derived-objects, component-objects, and type-objects:

- *Derived-objects* represent those objects that can be (re)constructed automatically from other objects.

- *Component-objects* represent those objects for which no automatic construction process exists (e.g., a "source"). In the literature these objects are sometimes referred

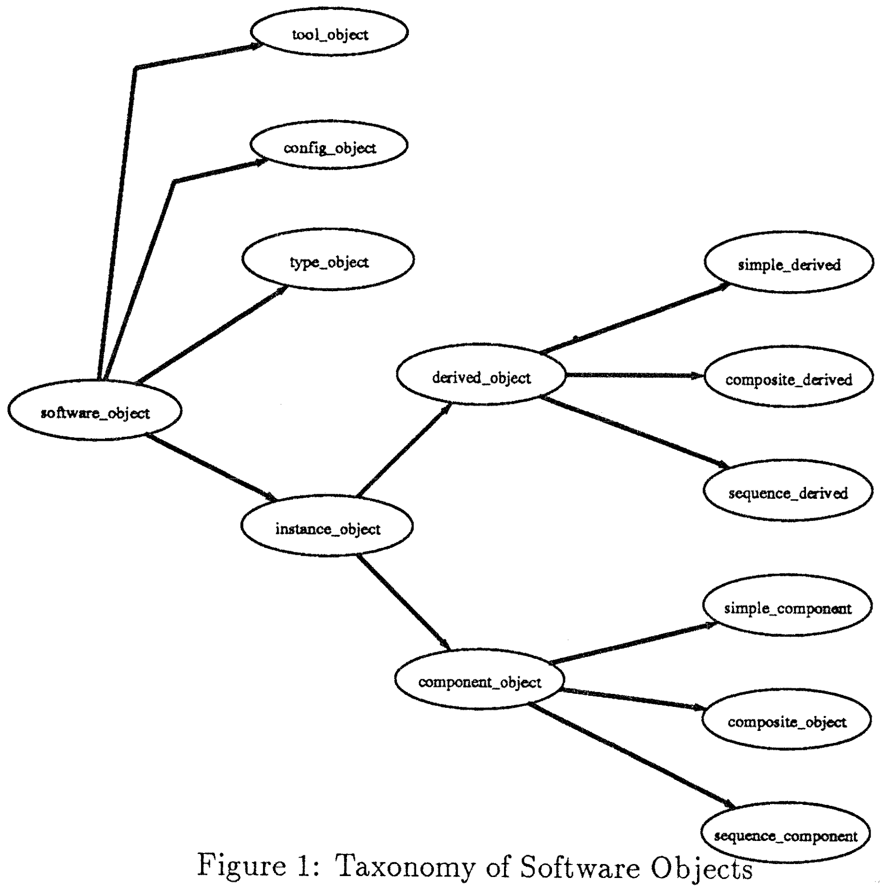

Figure 1: Taxonomy of Software Objects

to as being “non-re-derivable” or “primitive”. The creation of these objects involves some form of human interaction.

- *Type-objects* define schematas for classes of objects.

Attributes associate values with objects. All objects of a particular type share a common set of attributes. Attribute values are typically constrained to have a fixed domain and perhaps a limited range (e.g., domain= integer, range=1:10). The attribute values can be used to uniquely identify a software object. For example, the following objects are distinct because they differ in at least one attribute value.

> object(name=foo,type=c,attributes(system=“UNIX”))
> object(name=foo,type=c,attributes(system=”SYSTEM V”))

Edges represent relationships between objects. In general, these relationships are some variation of the *depends-on* relation. For example: the output of a tool *depends-on* its inputs and its parameters, or a module *depends-on* the functions it imports. One way of decomposing the configuration management process into sub-processes is to partition the *depends-on* relationship into sub-relationships. Of particular interest here are the sub-relations *is-component-of* and *derives*.

The *is-component-of* relation represents the use of one component by another component. Some concrete examples of this relation could be: includes, imports, calls, requires, or uses. In general this relation holds over specific objects, not classes of objects. For example, if module A uses (includes) header B, then B *is-a-component* of A (written *is-component-of*(B,A) [2]). Some systems define this relation statically using a structure such as a system model[Leb84]. Others define it dynamically by looking into the body of a module to determine which other modules are needed[Est85].

The *derives* relation represents the transformation of one object into another object. The *derives* relation holds over type-objects. An example of this relationship can be found between type-objects representing programs written in C (i.e., of type “.c” [3]) and their compiled counterparts (type “.o”). This might be written as *derives*(c,o).

A *configuration* is a formal specification of all the information needed to construct a desired product. Figure 2 shows an complete example of a configuration. It includes component-objects, derived-objects, tools and tool parameters, as well as version/variant and dependency information. The definition of what a *configuration* actually is, is a hotly

[2]Throughout the paper *relation(A,B)* can be interpreted as: *A relation B*.

[3]We use the typical Unix convention of denoting the type by the file name extension.

debated topic[Win88]. To some it is a description, to others it is an execution of a description. Others still suggest it is some combination of both. For the remainder of this paper references to a "configuration" imply only the formal specification.

The *Product Graph* is an internal graphical representation of a configuration. It represents the "working" graph throughout an entire transaction. From the time the user makes a request until the time the product is actually produced, this graph will be the focus for a series of transformations. Throughout the model, names will be given to several of the well-defined states of the product graph. The product graph, and its well-defined states, is a key to understanding the model.

2.1 The Knowledge Base

Configuration managers must rely on some form of database to store the various software products and configurations [Nes86,BL85,DL86,Ber87,Wie86,Cle86,Pen79,KC87,SCB86,AH87, Pen87,SZ87,MACK86]. Due to the importance of this repository, it is essential that it be represented in the configuration management model. The configuration management model must be general and flexible enough to describe various configuration management systems. Similarly, the model of the repository must also reflect these same characteristics.

For most operational systems, this database is the file system of the underlying operating system. Software objects are stored as files, and additional files are used to record properties about the software objects. Examples of these property files include Make's *Makefile*, and Odin's *ODIN* directory. DSEE[Leb84] relies on a modified file systems for object support. The Apollo file system directly supports difference files similar to those used by RCS [Tic82][4]. It also allows properties such as version numbers to be stored as part of the file system. The Adele system supports an attributed object store in which objects can have user defined attributes[Est85].

2.1.1 Knowledge Base Requirements

The database requirements for configuration management are similar to those needed for the more general Software Engineering Environments (SEEs). As Nestor[Nes86] pointed out, neither general purpose databases or file systems are appropriate for SEEs. Many researchers have tried to define an appropriate set of requirements[DL86,Ber87,HS88,BL85, MW87]. Unfortunately, the set of requirements defined by each of these researchers is different. One reason for this is the different database philosophies of the authors: do they

[4]As a result all tools must be modified to deal with the specialized internal representation.

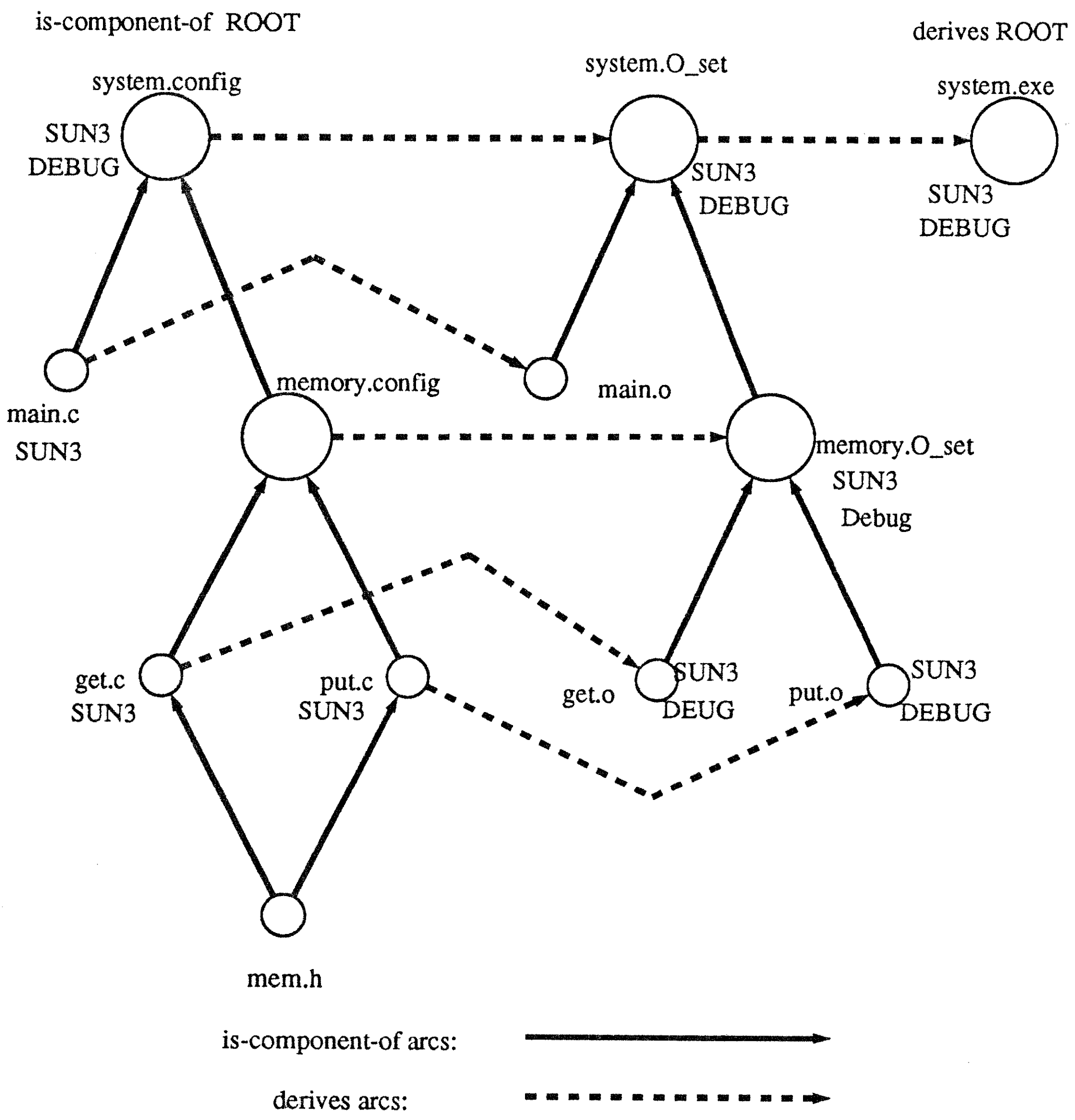

Figure 2: An example of a configuration

advocate the E-R model, graphs, or an object orientation as their representation? Another reason is that there is no clear definition of the configuration management process and therefore no clear definition of the requirements for its database. Despite this, there does seem to be a minimal set of requirements on which most people agree:

Supporting Flexible Data Types
Object Clustering
Attribute Support
Relation Support
Rule support
Version support
Consistency control
Multi-user support
Authorization and Access control
Transaction and Recovery Support

From this point forward the term "knowledge base" will be used to describe the persistent repository. This distinction is made to indicate that both data and rules are being stored. The inclusion of rules, support for triggers, and active attribute values allow for the possibility of some of the functionality of the configuration management model to be provided by the knowledge base. Whether the functions needed to support the above list of requirements belong in the knowledge base or in the configuration management model is an open question.

2.1.2 Knowledge Base Structure

Conceptually, it is convenient to view the knowledge base as a collection of functionally-distinct entities. This decomposition reflects inherent differences in the kind of information found within the knowledge base as well as the way in which this information is used. Some information is generic, some specific; some information can be recomputed and some can not. The knowledge base is an important part of the model. Understanding its structure will clarify the way in which the various entities it contains are used in the later sections of the paper.

When trying to model configuration management systems, it is convenient to partition the knowledge base by distinguishing between *instance* objects and *type objects.* Type objects describe class-level information. Instance objects describe unique instances of type objects. Type objects and instance objects are different: they are used at different times and in different ways, as explained below.

The *Type Graph* (TG) (which is stored in the knowledge base) is composed of type-objects, relations, and attributes. Attributes associated with type-objects are typically tools needed to derive that object, and any possible parameters that effect those tools.

Type-objects are similar to "frames" in conventional frame based models[Cle88c,Nar88]. Some of the characteristics type-objects and frames have in common are:

Types describe common characteristics of their instances
Types form a hierarchy along the sub-type/super-type relation
A set of common attributes is defined for all instances of a type
A set of common operations may be defined for all instances of a type
Attributes have restricted domains
Attribute definitions are inherited along sub-type links.

One characteristic that distinguishes the Type Graph from frame based representations is that the type-objects are connected to one another by edges of the *derives* relation and form a forest of graphs. The edges describe possible type-to-type manufacturing sequences. Because the *derives* edges form a graph it is possible for more than one manufacturing sequence to exist between two types.

Figure 3 represents a simple Type Graph. Using the Type Graph one can determine that it is possible to construct an o-object from either a c-object or a fortran-object. It is not possible to construct an o-object from an aladin-object using this Type Graph because no *derives* path exists between them.

The *Instance Graph* (IG) is also stored in the knowledge base. It contains instances of type-objects found in the Type Graph. It also contains all attributes associated with, and all the edges that form relations between those objects. All known object instances appear in the instance graph. [5] Attributes associated with instances form *(name,value)* pairs. The values of attributes may be simple values or complex functions. Relations between objects are represented as edges in the knowledge base. These edges are represented as tuples in the form: *relation(object,object)*. For simplicity, all relations are binary and directed. Both component-objects and derived-objects are found in the Instance Graph.

Although the Instance Graph and the Type Graph should be considered as separate entities, links between them do exist. For every instance of a particular type there must

[5]This is not completely true. As described in a later section all valid objects will exist in this graph only at commit points.

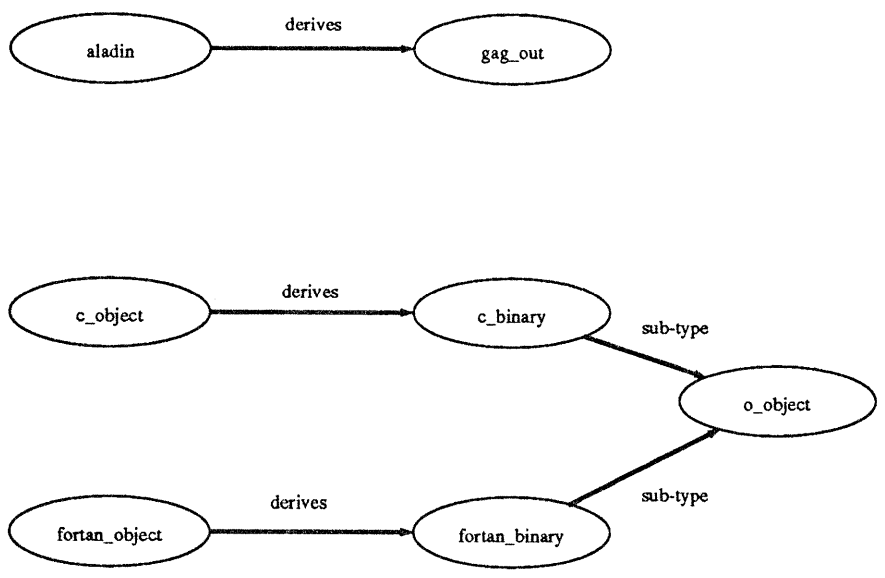

Figure 3: A simple Type Graph

exist an *instance-of* edge *instance_of(instance-object,type-object)* and its inverse relation *type_of.* The *instance_of* relation is needed to locate all instances of a particular type. The *type_of* relation allows for an instance-object to refer back to its type.

The knowledge base can also be functionally decomposed into a global view and possibly many local ones. The global view represents the shared knowledge base common to all users. The local views contain the same kind of information but only about the local objects each particular user is working on. In this way information about a transaction in progress can be insulated from all other user views until the local information is committed.

Most configuration management systems recognize the need for "checking out" [6] components, modifying them, testing them and checking them back in[Leb84,Tic82,M75]. For example, a user checks out a version of "foo.c" from the global knowledge base, makes changes to it, builds a product that depends on "foo.c" and then tests the product. Unless the user is perfect it may take several cycles of the modify/build/test cycle to convince the user that the changes are sound.

Until the user is ready to check the new instance of "foo.c" back into the global KB it is inappropriate to clutter the global KB with numerous, possibly inconsistent, intermediate

[6]The term "check out" is borrowed from RCS. It does not mean that the object disappears from the global KB view , rather that a copy is moved to the local KB view.

results. Logically, it would be better if these changes were restricted to a local view. When the user is ready to check in a new version of "foo.c" the appropriate information from the local view will be committed to the global knowledge base.

Another partitioning, and one that has received very little public exposure, is parallel structure for private information. This structure, or partition, is often referred to as a *cache* [7]. One approach to optimizing the configuration management process has been to keep side information about the previous configuration construction and execution processes. In existing systems, what information is kept and how that information is used has been completely up to the system's architects. This is understandable; a mistake in the logic of the cache manager could have profound effects on operations of the system.

In DSEE an object that describes the complete construction history (BCT) of each derived object in the object pool is stored to optimize reconstruction. In Odin this is done by a rather involved process which stores the derivation path, and thus the location of the derived objects[Cle88b]. In Wagner's work [Pfr86] version maps and attributed systems trees are kept in a database.

The determination of what information is to be cached should depend on the focus of the users of that system. In early stages in the life of a product, when the product's structure is not fixed, it is logical to cache information about the dynamic construction process. In later stages this information may no longer be useful it may be more logical to devote most of the cache to manufacturing process information.

This model makes an important distinction between the knowledge base and the product graph. Normally, the product graph is not considered part of the knowledge base. In theory, we may construct the same product graph each time a product is required. In order to optimize construction of the product, however, pieces of the product graph, or perhaps even the complete product graph, may be introduced into the parallel caching knowledge base. A reasonable mechanism for controlling such optimizations is a significant subject of research and will be dealt with throughout this paper.

2.2 Basic Graph Transformation Mechanisms

Two kinds of basic transformation mechanisms are used in this model to augment the product graph: *extension* and *attribution.* An extension transformation takes the product graph and adds nodes and edges. The nodes represent software objects and the edges represent a relation between these objects. Extension actually changes the shape of the

[7]In some systems the only information found in the cache is derived-objects, and therefore the cache is sometimes called the "derived object pool".

graph. An attribution transformation, on the other hand, decorates the product graph by setting the attribute values of the nodes, but does not change the shape of the graph.

These basic transformations are important because they characterize the phases of the model. The *relationship being considered* will be the term used when describing the particular relation that will govern the transformation process. Extension and attribution are always performed with respect to the relation being considered. For example, one phase in the model (which produces the Component Product Graph) is characterized by extension and attribution with respect to the *is-component-of* relation.

Where possible, attribution transformations will be described separately from extension processes. In some cases the two may need to be intertwined, since complex extension processes may require attribute information in order to decide what edges to add to the product graph.

2.2.1 Extension

Extension is the process of adding nodes, and edges of the relation being considered to the product graph. In its most basic form, it is little more than a specialized inferencing mechanism. The inputs to the extension mechanism will be facts, rules and and inference parameters. The facts and rules reside in the knowledge base.

There may be several extension transformations during the configuration management process. In any one of these transformations, extension will only take place with respect to the relation being considered. All other relations will be ignored during this particular extension process. Each extension process is guided by a set of rules. Many extension rules may be defined for any specific process. One major constraint on the extension process is that every node in the graph will be visited once and only once.

In the simplest form of extension, specific information that describes the entire process exists in the knowledge base. For example, assume the following facts are in the Instance Graph:

derives(test.c,test.o)
derives(test.o,test.exe)

If we start with the node *test.c* in the product graph we can transitively determine that test.exe can be derived from test.c. To get a better understanding of the basic mechanism one could examine the process in detail and see its effects on the product graph. Using the default extension process outlined in Figure 4, to satisfy step (1) R is set to the *derives* relation and to satisfy step (2) current is set to the node *test.c*. Then, via step (3), it will be

```
(1) let R be the relation being considered
(2) Set current to the root of R.
(3) For all X in the KB such that R(current,X) is true
      If X is not a node in the product graph then
            Add X to the Graph
            Add edge R(current,X) to the graph
(4) Using a breadth first left to right search sequence
visit each node, set current= new node and repeat (3)
```

Figure 4: The Default Extension Process

determined that the edge *derives(test.o,test.c)* satisfies the condition and therefore the node *test.o* and the edge *derives(test.o,test.c)* will be added to the product graph. According to step (4) we set current to *test.o* and repeat (3) adding the node *test.exe* and the edge *derives(test.exe,test,o)*. At this point, there are no other facts that will extend the graph. The result of the extension process is an updated product graph in which all possible nodes and *derives* edges have been added.

A more concrete example is shown in Figure 2. In this case the default extension transformation, with respect to the *is-component-of* relation and starting with the node *system.config*, will add the nodes *main.c* and *memory.config*, and their connecting edges. It would then add *get.c* and *put.c*. Finally, *mem.h* would also be added to the graph.

The default process is potentially ambiguous since there may be many out edges that originate at a given node. Ambiguity is resolved by forcing the extension to occur using a breadth-first left-to-right consideration of the nodes, and stopping when no further traversal is possible.

A slightly more complex extension process, as discussed in detail in later sections, is the case where the *derives* information exists between type-objects. For this process derivation relations between instance-objects must be inferred from type-object information.

One can imagine even more complex extension processes in which the result of one extension dynamically controls the remaining extension process. For example, assume that the system being modeled supports multiple versions of the tools to be used in the construction process. Furthermore, assume that the inclusion of one tool excludes the possibility of another. In this case the extension process will be controlled by the constraints between the tools.

2.2.2 Attribution

Attribution is the process of setting the values of attributes in the product graph. As with the extension process, attribution is carried out with respect to the relation being considered.

The sequence of attribute computations is determined by the dependencies between the attributes. Some attribute values will be initially set; others will have to be computed. If an attribute depends on no other attribute values then its value can be set at any time. If the value of an attribute depends on one or more other attribute values then it can be set any time after the values of the attributes on which it depends are set.

The basic attribution process is derived from two different notions of inheritance: *structural inheritance* and *value inheritance. Structural inheritance* describes **what** attributes are inherited by an object. Structural inheritance closely resembles the notion of inheritance found in the object-oriented domain. When an object is created it "inherits" attributes from its type and all of it's super-types. Structural inheritance is used to determine the set of attributes associated with a particular object. It may also describe the domain and range of possible attribute values and even possibly a default value. What it does not describe is how the values of the attribute will be determined. For example, assume we have the object classes *software-object* and *component-object*, as in the taxonomy found in Figure 1. In this taxonomy *software-object* is the super-type of *component-object.* Associated with any *software-object* are the attributes **name** and **type**. Associated with any *component-object* are the attributes **ct** and **version**. In the process of creating an instance of a *component-object*, structural inheritance will define the set of attributes associated with this instance to be: **name,type,ct,version.**

Value inheritance describes **how** attribute values are set. The notion of value inheritance is borrowed from the attribute grammar domain. It describes the various interdependencies between the attributes in a declarative way. In the case where the value of an attribute depends on no explicitly stated dependencies, there is an implicit rule of value inheritance. The implicit rule states that the value of an attribute at a node is defined to have the same value as its parent.

Figure 5 represents a simplified state of a product graph. In this graph the root (test) has the initial value for its **ct** attribute to the rule given in Figure 6. Figure 7 is a description of all the dependencies between the attributes in a compound-object. Note that there is no mention of how the value of the **ct** attribute is set. In this case, the default attribution process supplies a rule that states that the value of **ct** in a node is inherited from its parent. As a result the attribute value "If name=STG then version=3 else version =4" is propagated from the root to all of its descendants. Once the value of the "ct" attribute is

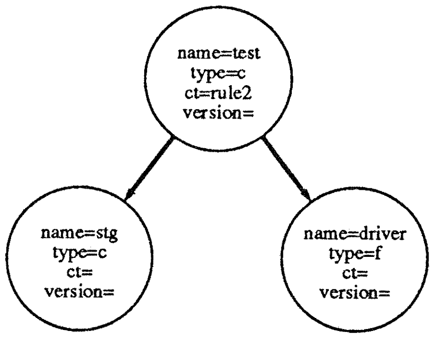

Figure 5: A simple incomplete Product Graph

```
rule2 = If name = "stg"
      then version = 5
      else version = 4
```

Figure 6: Initial value of "ct" attribute

```
xC-Obj.version ::=
gen_version(xC-Obj.ct,xC-Obj.name)
```

Figure 7: Attribution rules

set at a node, it is then possible to determine the value of the "version" attribute. This is done by applying the **gen_version** function to the attribute values at that node.

In the above example, structural inheritance was used to determine what the attributes of an object are and value inheritance was used to determine how the values are to be set. Where no rules are supplied for setting the value of an attribute, the default attribution mechanism states that the value is to be inherited from its parent. It is worth noting here that the rules for setting attribute values, both implicit and explicit, are declarative in nature. They describe how to set the values but not when. It is the job of the system, not the user, to worry about the order in which attributes are set.

The characteristics of the attribution process of the various phases of the configuration management process differ in the kinds of attributes being propagated. In some phases, attributes will have few inter-dependencies and can be evaluated efficiently. In others, the attribution may be very complex. Attributes may depend on other attributes in such a way that the order in which their attributes are evaluated can not be determined a-priori.

In some cases the attribution and extension processes may have to be intertwined. For example, if an extension process depends on attribute values, and attribute values depend on the extension process, then alternating extension and attribution is needed.

3 The Model

The model described below captures the configuration management process that starts with a user request and ends when the request is satisfied. Configuration management is explained in terms of the product graph, the transformations it goes through, and finally its execution. The process is decomposed into two major phases:

- **Graph Construction:** Starting with a very high level user specification, build up a legal and consistent configuration.

- **Graph Execution:** Once constructed, the configuration graph is used to drive the execution process. The manufacturing steps are executed, the derived-objects are produced, and the product is made available.

The graph construction phase, through an ordered series of well defined transformations, takes the initial request and builds up a complete configuration. Recall that a complete configuration is comprised of all the objects and their attributes, dependency information, and manufacturing steps needed to fully describe how to construct the desired object(s).

Configuration execution is a complex, time consuming task. In order to efficiently execute a configuration, optimization and scheduling techniques are introduced. In many cases optimization, scheduling and execution might be heavily intertwined.

3.1 Graph Construction

The graph construction phase is characterized by the building up a configuration from the initial request. This phase is further decomposed into the following distinct tasks:

- Transform initial request to product graph representation
- Augment product graph with manufacturing sequence
- Augment product graph with set of primitive objects needed to drive the manufacturing process
- Augment the product graph with the necessary intermediate derived-objects.

Four distinct states in the construction of a complete configuration are outlined in the following sections. Each state is given a name based on what new information has been added to the product graph since the previous state.

3.1.1 Partial Product Graph

The *Partial Product Graph* (PPG) represents an initial, very high level, specification of the desired product. The configuration management process begins when a user makes a request for a particular product to the system. The first step in transforming this specification into a complete configuration is to create the Partial Product Graph. The creation of this graph is a straightforward transformation of the user's request into an internal representation.

The important characteristics of the PPG are its relation *roots* and the set of initial *attributes.* The two relations, as you may recall, are *derives* and *is-component-of.* The root of the *is-component* relation describes the initial primitive object from which the product is to be produced. All of the initial attributes are attached to it. The *derives* root describes the type of product desired.

An example of a simple request could be:

from the specific object "system.config" produce the product of type "executable".

In this example the component-object "system.config" defines the root of the *is-component-of* relation. "Executable" in this request specifies a type-object. This type object defines the root of the *derives* relation.

An example request for which there are initial attributes for the *derives* root could be:

> *from the specific object "system.config" produce the product of type "executable" in which all derived objects are built with the debug option.*

In this case the initial attribute value for "tool parameter" would be set to "debug".

An example of a request for which there are initial attributes for the *is-component-of* root could be:

> *from the specific object "system.config" produce the product of type "executable" in which all of the components that make up the system are from version 5.*

In this case the initial value of the "version" attribute would be set to "5".

Of course one can imagine much more complex forms of attribution then described above. In this model, attribute values can take the form of simple values, a set of rules, or even arbitrary functions. To produce the PPG, the initial values (no matter how complex) are associated with the roots of the relations; nothing more.

3.1.2 Typed Product Graph

The Typed Product Graph (TPG) represents the addition of a legal construction (manufacturing) sequence between the roots of the partial product graph. The process that creates the TPG addresses the question: *Does a consistent sequence of manufacturing steps exist to create the desired product from the initial object?*

The TPG is produced from the Partial Product Graph by extension and attribution transformations with respect to the *derives* relation. The result of the extension process is the inclusion of type-objects and *derives* edges from the Type Graph into the product graph. The edges are directed and represent an ordered manufacturing sequence that starts with the type of the initial object and ends with the type of the product. The nodes (type-objects) and edges do not describe all of the details of the manufacturing process. Rather, they describe a template whose details will be filled in by later transformations.

The attribution process decorates the newly augmented product graph with attribute values such as: names, tools, tool versions, and tool parameters. This process can be viewed as further refining the generic type-object manufacturing specification above into a more concrete one.

In most cases the extension and attribution processes are simple. The default extension process described elsewhere is used to infer nodes and edges from the Type Graph. The default attribution process is then used to propagate initial attributes from the *is-component-of* root to all of the nodes. These initial attributes primarily come from the user request. A typical example of initial attributes that are propagated here are tool parameters.

In many systems the Type Graph will support the notion of tool-type equivalence. Tool-Type equivalence states that a tool exists for, and is associated with, each type-object. An elegant property of this notion is that discussions of tools are abstracted away: by mentioning the type-object one can infer the appropriate tool. Odin's derivation graph and, to some extent, Make derivatives and DSEE make use of tool-type equivalence.

Complications arise when versions of tools are introduced. Adhering to the notion of tool-type equivalence implies the introduction of a new type for every new version of a tool. In a large, long lived environment with many tools, tool-type equivalence may become unmanageable. Perhaps the best way to understand the implications of tool-type equivalence is by example.

In the examples below two new versions of a compiler are introduced. In first example the new compiler is decomposed into two tool fragments that share an intermediate representation (produce an intermediate object). In the second example the new compiler is a version of the old in which the output is incompatible with the original compiler's output.

Figure 8 represents a Type Graph before either of the new compilers are introduced. It supports tool-type equivalence and is shown here as a base reference point. Figure 9 and Figure10 are Type Graphs that represent two possible ways of describing the old compiler and the two new compilers. In Figure 9 tool-type equivalence is maintained. Figure 10 attempts minimize the proliferation of types by introducing a more general notion of tool types.

Example 1: In this example the new compiler is decomposed into two fragments (P1 and P2) that share an intermediate object. The introduction of a new object presents problems for both TGs. In Figure 9 new type-objects (and some edges) must be introduced to represent the new tool fragments. Moreover *is-a* edges must be created to coerce the binary of the old compiler and the binary of the phased compiler to a common type [8].

The introduction of the new phased compiler (type-objects) also has an impact on manufacturing sequence selection process. Because of the intermediate object the length of the new path to the desired product will be longer than the original one. Recall that the default extension process uses breadth-first left-to-right search to find the path. Breadth-

[8]The Coercion is necessary if the two types are to share a common construction sequence fragment.

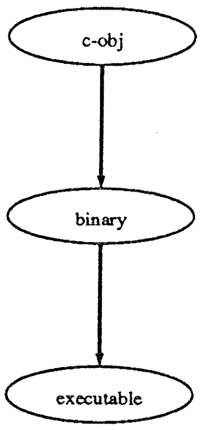

Figure 8: Original Type Graph before new compilers

first search guarantees that the shortest path will always be selected. Three possible ways to override the default mechanism are:

1. Introduce a global constraint that always selects the new compiler.
2. Provide a mechanism in the request language for selecting alternative paths.
3. Introduce a policy specification that locally constrains the extension process.

In Figure 10 the tool-type equivalence is broken. The type-objects in the Type Graph are more general and refer to a general tool-object. By using this more general schema the selection of the construction sequence from the Type Graph is simplified while the tool selection process becomes more difficult.

Example 2: In this example a new version of the compiler is introduced that has the same input/output semantics as the original compiler. The problem here is that the use of the new compiler demands the use of the new binder and visa versa. It turns out that the new compiler differs from the old in because the new binary objects contain additional bind information. For this reason, it is necessary for the new binder to be used with the new version of the compiler.

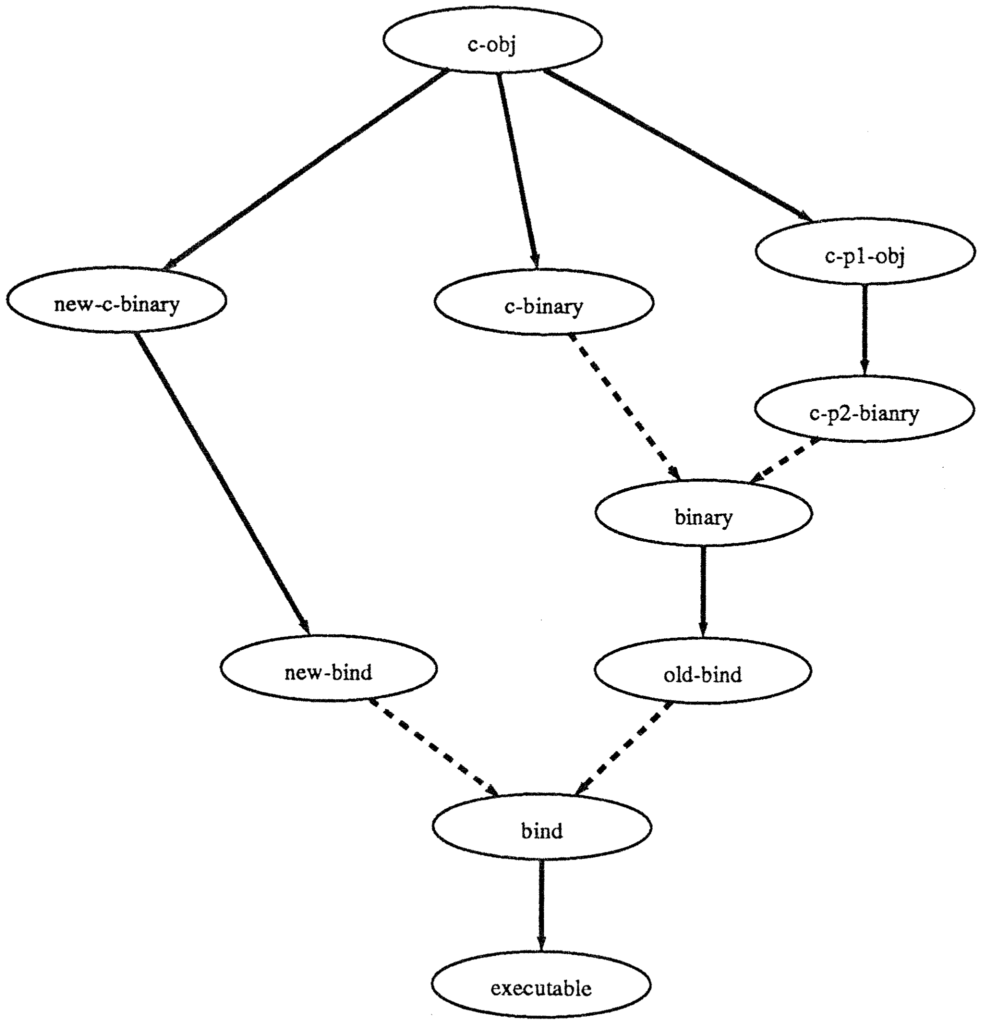

Figure 9: Type Graph with tool-type equivalence

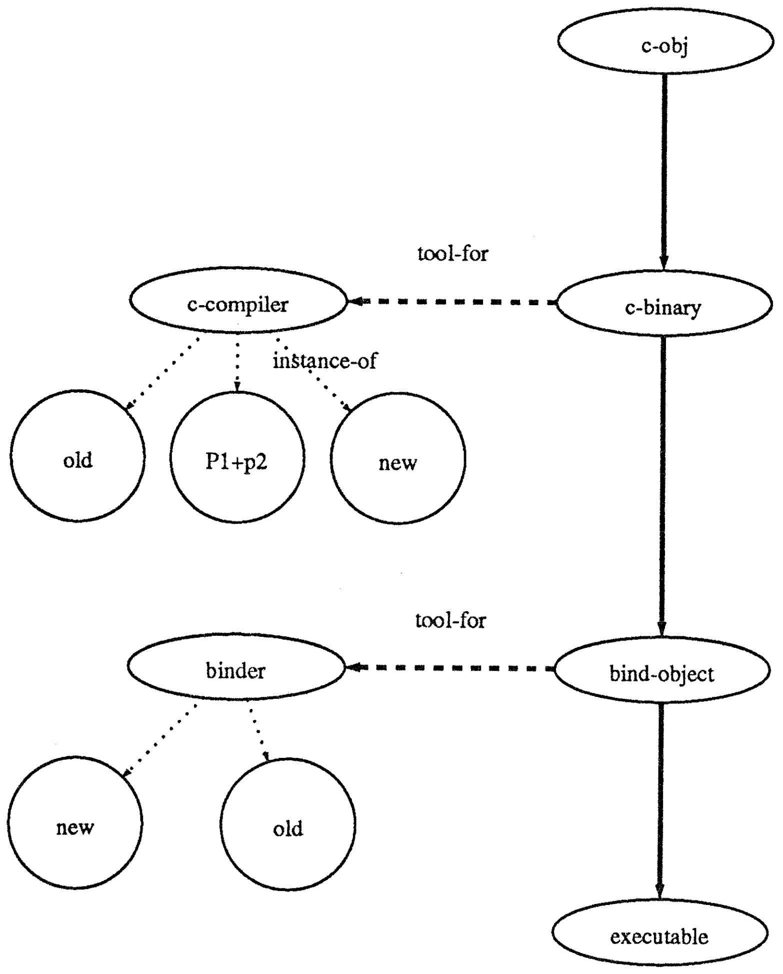

Figure 10: Type Graph with minimal types

If the new compiler is requested in a transaction, it is clear that the constructed manufacturing sequence must include the new binder, not the old. In order to guarantee that the new binder is selected, it is essential that a constraint exist which guides the tool selection process. In the tool-type equivalence scenario (Figure 9) the constraint is encoded in the edge. In Figure 10 the constraint is described as a fact in the knowledge base.

Implicit constraints are not desirable. They overload the semantics of the edges and promote type proliferation. Assume for a moment that a Type Graph such as Figure 9 exists in which the constraint between the new compiler and new binder is implicitly encoded in the edge. Imagine a scenario in which a very long manufacturing sequence which starts the executable type and then the graph fans out again. If the reason for the fan out is because of the new binder, for example, it may be that the entire common sequence may have to be duplicated.

The concept of using types and typing is an integral component in the construction of the manufacturing sequence described in Type Product Graph. The concept is not entirely new and in part is borrowed from existing systems. The remainder of this section serves as a summary of how and to what extent existing systems provide for the construction of manufacturing sequences.

Make (or at least Make derivatives: see[Fel88]) have a limited capability for performing this extension process. In Make the user is allowed to specify default rules[Fel79]. These are class rules that describe generic transformations from type to type. The way in which these rules are used is rather limited. Make systems in general perform a depth-first rather than breadth-first search. This makes it impossible to find alternative paths. In some Make systems the number of default rules that can be applied in an unbroken sequence may be very limited[9].

The Odin system has a well-defined type system[Cle88a] that supports the kind of processing described above. In Odin it is possible for there to be more than one path between the source and product types. It uses the default extension process described earlier in which breadth first search causes the selection of the shortest path. To override this it is possible for the user to specify articulation points[Cle88b] that force the extension mechanism to select alternative paths.

In other systems, type rules either do not exist or are simply used as place holders. In Adele (at least in the published documentation) there is no mention of type objects. In DSEE type objects exist but are not used in the extension process; they are simply attribute place holders.

[9]The original Make allowed only one default step in a sequence

3.1.3 Component Product Graph

The *Component Product Graph* (CPG) represents the augmentation of the product graph with the complete, fully attributed set of primitive components needed to build the desired product. The Component Product Graph is produced by applying extension and attribution transformations, with respect to the *is-component-of* relation, to the product graph.

The previous transformation determined the existence of a legal and consistent manufacturing sequence to build the desired product. The goal of the following transformations is to determine if there exists a legal and consistent set of primitive components that will drive that manufacturing process. For many systems the components are objects such as source modules, include modules, etctera. Selecting the set of components involves finding out the dependencies between objects, selecting the correct version/variant of those objects and possibly performing some semantic analysis to determine whether the selected set of objects are consistent with respect to one another.

The construction of the component set, or some similar structure, is a very active area of research within the configuration management community[Est85,Per87,MK88,Wie88, WF88]. In early CM systems the main emphasis in constructing this set of components was centered on dependency issues[Fel79]. The next stage of evolution was to include pragmatic constraints[Est85,Leb84,BLLV87] in which the user could define various constraints about version/variant selection: e.g., *I want version 4 of a particular system.* In more recent systems emphasis is being placed on semantic constraints[WF88,Per87,MK88,Tic86,HKM87]: e.g. *Is there an export for every import.*

In terms of this model, early CM systems dealt exclusively with the simple (static) extension processes. The next stage of evolution was to augment the simple extension process with dynamic version/variant attribution. After a time certain system designers realized that a more dynamic extension process was also needed: rather than apply version/variant attribution to a statically defined structure, this attribution was used to dynamically determine what other components were needed. Other systems investigated other orthagonol forms of attribution, in particular semantic analysis.

In this model it is possible to define any of the above forms of attribution. In particular it is possible to perform both version/variant selection and pragmatic analysis within the same system. It is worth noting that the evolution in this particular aspect of configuration management is towards supporting greater and greater complexity and dynamism within the model. The hope here is that all this extra, automated, complexity will result in the generation of more reliable software products.

As in the previous section the transformations that take place in this section are extension followed by attribution. The significant difference here is that these transformations are done with respect to the *is-component-of* relation. The extension process will add nodes and edges of the *is-component-of* relation to the product graph. The attribution transformation in turn will set the attribute values for these newly added nodes.

In Make the Makefile statically defines the *is-component-of* relation. The extension process is basically a transitive closure on the *is-component-of* relation; there is no built-in mechanism for performing attribution.

In DSEE the extension process is even simpler than in Make. What DSEE does that Make does not is to provide a version/variant attribution mechanism. The DSEE Configuration Thread (CT) describes the attribution rules to be used in performing version selection. These rules can take the form:

```
for ?* -when RESERVED -use_options -debug
for ?* REV9_BUGFIXES[] -when_exists
for ?* []
```

The above three rules represent a complete CT. The rules are applied in sequential order at every node until there is a match. The first rule states: if a file is checked out (RESERVED) then use the current working copy and compile it with the debug option. The second rule states: if the component exists as part of the REV9_BUGFIXES, use it. The third rule states: use the most recent version from the version library.

In the Adele system the attribution transformation aides in driving the extension transformation. In general the overall process in Adele can be thought of as: extending by one node, attributing that node, and then using the results of that attribution for the next extension. This cycle continues until there is no further node to add.

Systems such as Adele and DSEE address pragmatic consistency issues in the selection of a set of source objects. They address questions such as:

Is the component set complete?
Does it satisfy the user's request?
Does it violate any user's constraints?

However they say nothing of semantic consistency. Many CM systems today are being built around one language.[10] For these systems the following issues need to be addressed as well:

[10] Or at least principally around one language.

For every import is there an export?
For every import/export pair are the number and kind of parameters consistent?
Does a version of a module body satisfy a version of a module interface?

Tichy[Tic86], Kennedy[HKM87], Perry[Per87] and Wagner[Pfr86] have noted that the use of semantic information aids in producing more precise products. They also suggest that this information can be used at later phases in minimizing the (re-)computations necessary to build the product.

It is important that the extension and attribution process in this and all other phases be programmable. As we survey current configuration management systems we can see an evolution towards allowing more and more programmability in the attribution process. With the extension process though, only recently has there been any work done which focuses on dynamic processing. Presently, there are no systems that allow for dynamic, programmable extensions along the *is-component-of* relation.

3.1.4 Product Configuration Graph

The *Product Configuration Graph* (PCG) represents the inclusion of all the necessary derived objects and their associated attributes into the product graph. The *Product Configuration Graph* represents a complete configuration. It contains all the source objects, derived objects and the tools (and possibly their parameters) involved in the construction process. The Product Configuration graph is constructed by applying extension and attribution over the *derives* relation. Most if not all of the information needed to construct this graph is already contained in the graph itself. Whereas the derives information needed to construct the Type Product Graph came from the Type Graph, the derives information for this process comes from the Type Product Graph. Again, the extension process will add nodes and edges, and the attribution process will propagate attribute values over the extended graph.

The extension process used to create this graph may be complicated by compound or aggregate objects. If a system supports sets of objects then that system must supply tools that support sets. The Odin system supplies such a set of tools. One of the system tools available in Odin is called "homomorphism". The homomorphism tool allows the author to apply the same derivation step to all the elements in the set.

Below is a fragment from an Odin specification. Of particular interest is the **o_list** class. It takes as inputs a compound file of "c" source files and produces a list of "o" objects. It also specifies how to build one from the other.

```
o_list(o) "list of o objects"
        HOMOMORPHISM (:o)
            : cmpd

cmpd(c) "List of C files "
        COMPOUND
            : ref

o "C object module"
        USER c_to_o.cmd
            : c
```

Perhaps the easiest way to describe the extension process is to to see an example in pictorial format. Figure 11 is the product graph just after the completion of the CPG. Contained within this graph is all the information needed to produce the product configuration graph. The tool attribute of the **cmpd** derived object tells us that we must apply the "o" derivation to all each of the elements of the ref object.

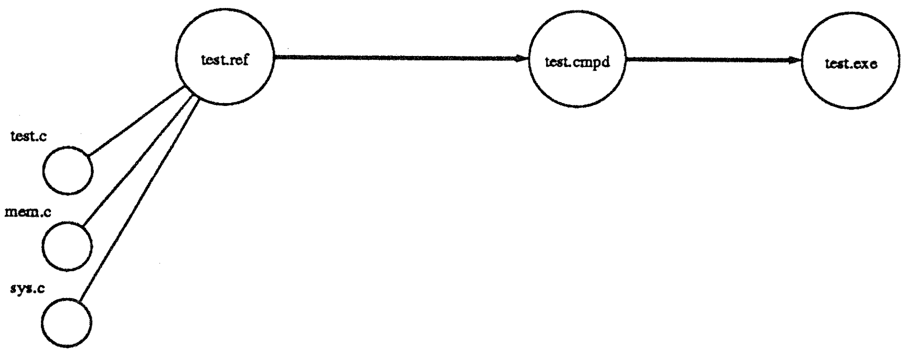

Figure 11: Product Graph before the addition of derived-objects

Figure 12 represents the expansion of the CPG to include the appropriate derived objects. The application of the homomorphism with respect to the "o" derivation generates for each element in the ref file an "o" object.

Once the graph has been extended to include derived objects the appropriate attribute values must be propagated. Of particular importance is the propagation of the attributes

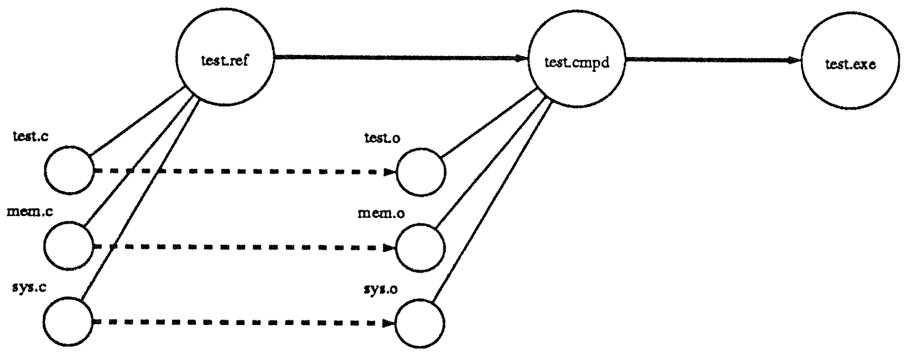

Figure 12: Product Graph after the addition of derived-objects

values for tool and tool parameters. The information used in the attribution transformation is gleaned from two sources: the generic attributes found in the Type Product Graph and the user request. From the TPG the tool attributes are extracted. This is true in Odin, Make, and DSEE. From the user request information about the parameters for those tools is extracted. At this point the construction process is finished and the result is a complete configuration.

3.2 Configuration Execution

After introducing a simple execution model optimization and scheduling techniques are discussed. The purpose of the optimizations is to eliminate any unnecessary or redundant manufacturing steps. Scheduling determines the best possible order of execution, given a fixed set of resources. It is worth noting that execution, optimization and scheduling activities may be heavily intertwined.

If you recall, a configuration is a graph in which the nodes represent software objects, attribute values parameterize the manufacturing steps, and the edges represent dependency information that constrains the manufacturing order. Execution of a node in a configuration can be interpreted as the application of a manufacturing step to a set of inputs. The output(s) produced by this application constitute the value of the software object.

In this model a manufacturing step is described by attribute values associated with the node to be executed. The tools, tool parameters and any other needed information are expressed as attribute values. This seems a natural way in which to describe the

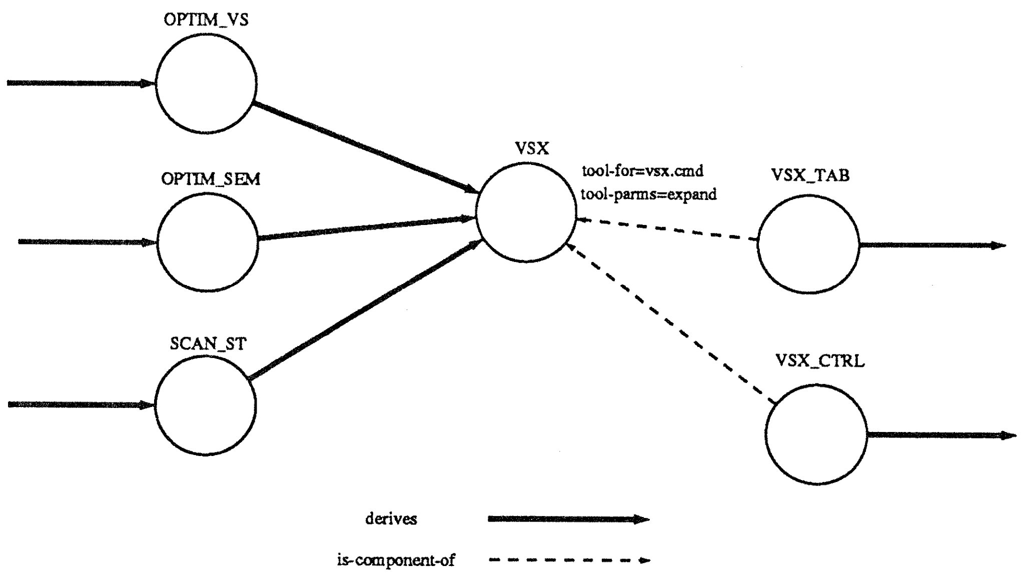

Figure 13: An execution model example

manufacturing step because the information is exactly where it is needed for subsequent recall: attached to the derived-object.

The set of inputs needed to drive the manufacturing step is defined to be the set of objects whose *derives* edges terminate at the node to be executed. Unless there is an error in the Type Graph, the set of incoming edges precisely describes the input set.

The output(s) resulting from the application of the manufacturing step is associated with the software object. In simple cases where there is only one output, the *value* of the software object is the output of the manufacturing step. In cases where there are multiple outputs, the value of the software object is references to other software objects.

A general property of the graph is that no node can be executed until all of the nodes it depends on have been executed. In effect, the dependency edges define a partial ordering on the sequence in which nodes can be executed.

Figure 13 represents a small fragment of a complex manufacturing process used to generate compilers in the Eli system [WHK88]. The manufacturing step describes the application of a tool fragment from the GAG tool set[KHZ82] that transforms an optimized attribute visit sequence into a visit sequence table.

The manufacturing step in this example is described by the **tool-for** and **tool-parms** attributes. The "expand" tool parameter plus the "vsx.cmd" tool are used to manufacture the vsx object.

The set of input objects that drive this manufacturing step are: **optim_vs, optim_sem** and **scan_st. optim_vs** represents an optimized attribute visit sequence, and **optim_sem** an associated set of attribute dependencies. Both **optim_vs** and **optim_sem** are produced by by a previously executed optimization manufacturing step. **scan_st** represents an identifier table produced by a lexical analysis tool.

The application of this manufacturing step produces two outputs: **vsx_tab**, a visit sequence table and **vsx_ctrl**, the control function that is driven by the table. As you can see in the figure, because there is more than one output, the two objects are represented as separate entities and are associated with the vsx object.

Before the vsx manufacturing step can be executed all of the input objects it depends on must exist and be "up-to-date". Similarly, subsequent manufacturing steps that depend on **vsx_ctrl** can not be executed until the vsx manufacturing step itself is successfully completed.

Of course configuration execution is not that simple. It can be complicated by a number of factors:

1. At the time of execution, some number of the objects will already exist in the database and need not be re-derived.

2. Some objects exist but are "out-of-date" with respect to the components from which they are constructed.

3. The execution may be terminated early if it can be determined that further execution will result in no changes to the database.

4. Some objects can be derived in parallel with other objects.

In most existing systems, execution, scheduling and optimization are heavily intertwined. Some systems use optimization techniques in the construction process and never create a complete configuration. Others may have complete configurations stored and completely bypass the construction phase. Another set of systems might perform scheduling, then optimization, before execution; others may optimize then schedule. Even heavier intertwining can be found in systems with dynamic optimization. In these systems the result of each execution step is used as the input to subsequent optimization and scheduling. Before discussing optimization and scheduling in detail, four examples from existing systems are given to suggest the flavor of what is possible.

In Make, a complete configuration, as defined in this model, is never constructed. Make uses static optimization techniques during the construction process to produce a reduced configuration. Each time a make request is issued, the Makefile must be parsed and a configuration is constructed.

In pmake, a parallel version of Make, the reduced configuration is constructed as in make. pmake's distinguishing characteristic is that it then schedules the configuration before executing it. Again, for each make request the make file must be parsed, the reduced configuration constructed and the schedule built before any execution.

In DSEE the *system model* and the *configuration thread* are "loaded". The loading of these two specifications is similar to constructing a complete in-memory configuration. When a *build* request is issued the complete configuration, (being constructed and in memory), is simply executed. When a change is made to an object in the configuration and a new build request is issued, the complete configuration is consulted and a reduced graph is produced. In a newer parallel version of DSEE, before the first execution a schedule is produced once from the complete configuration. All subsequent executions are determined from that schedule.

In Odin optimizations take place during execution. After executing a manufacturing step it is possible to determine whether the result is identical to a previous execution by comparing it to back level instance of the object. If the results are identical all subsequent manufacturing steps that depend on the newly created object are unnecessary and can be eliminated.

3.2.1 Optimization

Execution of a node establishes a constraint between the supporting objects and the constructed object. This constraint says that the constructed object has the value that would be obtained by applying the tool to the supporting objects. This constraint specifically does not require that the tool actually be applied every time the output object is needed. This is the key to optimizing the execution of the graph as a whole. If by some means it can be determined that the application of the tool would not produce a result that is different than the current value of an object, then the tool need not be applied at all. Borison [Bor86] has formalized some aspects of this in her *difference predicates.*

A number of means exist for determining if the constraint may (or may not) have been violated:

Timestamps: If the time of last modification of an output object is earlier than the times of any of its supporting objects, then it may imply that the constraint is violated.

Value equivalence: Apply the tool and if the result is bit-for-bit identical with the existing value, then it can be assumed (by construction) that the constraint was not violated.

These tests are not of much use individually, but they are useful in conjunction with the following rule:

Input equivalence: If none of the supporting objects has violated its constraint, then the dependent object may be assumed to be in compliance with its constraint.

This rule is important because it allows execution to cease at any point if the constraint is known to hold for all un-executed objects. As the graph is executed, various nodes may be marked as executed even if no work is done; they need only satisfy the constraint. The effect is that changes to source objects will propagate only to those objects that directly are affected by those changes.

Imagine for a moment a long, complicated construction sequence that starts with a compiler source object. In systems that do not support value-equivalence it is possible that the entire construction process might have to be executed. If, on the other hand, value-equivalence was in force and the result of the compilation was identical to that of the previous compilation then the previous object could be used in its place. Assuming that the entire construction sequence exists for the older copy of the input, the construction process can be terminated at this point.

3.2.2 Scheduling

Graph scheduling is concerned primarily with resource allocation and execution order determination. Factors that are used to determine a schedule include: dependency information, number and kind of resources, resource location, and possibly statistics about previous executions.

With the advent of powerful workstations there has been a shift from central to distributed processing. In such environments there are many machines that sit idle for significant periods of time. Devising configuration management systems that take advantage of these idle resources is desirable.

Distributed computations pose many problems for scheduling. For homogeneous environments some problems are:

1. Certain resources exist only on specific machines.

2. Machines may have different computing power.

3. There must be a guarantee sufficiently compatible execution environments

Configurations contain tool information. This information may place additional constraints on the execution schedule. For example, in an environment in which there is only one licensed Ada compiler, all Ada compilations must be scheduled for the machine that has the license.

In cases where it is known that a manufacturing step is computationally expensive, and there are many processors available, it is desirable to schedule that step's execution on the fastest available machine. Similarly, if several ready tasks have the same execution time but one has more out edges, (objects which depend on it) it should also be scheduled on the fastest available machine.

For a computation to be executed on a foreign machine, some guarantee must be made that the environment in which the actual computation is performed is sufficiently similar to that on the local machine. For example, the version and location of tools should be identical. DSEE is able to do this by a using a facility contained in the operating system. The Apollo operating system provides a facility to change file system root of a remote process to that of it's creating process. Of course there is an overhead cost associated with this. The overhead for starting an execution is the limiting factor for single threaded scheduling. In DSEE the degree of parallelism is roughly computed as: (the average of all execution times)/execution startup. The overhead associated with remote process startup in the Apollo operating system is on the order of seconds. The average execution time for a manufacturing step is typically less then one minute. Because of this, DSEE systems typically see a factor of 5 speed-up[LCS88]. There are several systems that perform parallel builds over distributed networks. DSEE[Leb84], pmake[Baa88] and nmake[Fow85] are all examples of existing systems.

Scheduling a set of independent tasks is NP-complete. Fortunately, heuristics have been found to produce feasible schedules. Some of the known heuristics for the *independent task-scheduling* problem are the list-scheduling algorithm, LPT (least processing time) and MULTIFIT[GGJ78]. These three scheduling heuristics distinguished from one another by the amount of information they have available to manipulate and how good schedule has to be. In general the more information available the better the schedule. In order to use MULTIFIT, additional information about the execution task time is needed. This implies knowledge either about the tools to be used (their speed, and the size of the input) or cached statistical information about previous executions.

4 Related Work

The purpose of this section is to mention related work and specific papers which have indirectly contributed to the proposal. These papers are not described in any detail. They are placed here to suggest to the committee the focus and breadth of the background research. If a member of the committee would like further information on any aspect of this section it will be possible to discuss this during the oral presentation.

Configuration management addresses a subset of the issues associated with more general Software Development Environments (SDEs). A primary difference between CMEs and SDEs is that SDEs try to address the entire software life cycle, from initial requirements to delivery and maintenance, in one integrated environment. Another significant difference is that in CMEs *everything* can be automated. SDE research is still in its infancy. This is obvious when one surveys the literature[Hen88]. One notable software environment project is Arcadia[TO88]. Arcadia is based on the notion of process programming[Ost86]. A sub-project under Arcadia is to define the process programming language APPL/A[HSO87].

In the model, extension and attribution transformations play an important role. Attribution as described in the model is based on attribute grammars[Rai80]. Attribute grammars and tools based on attribute grammars are areas of active research[KHZ82,sKK86].

One of the issues to be addressed in the thesis is whether or not the attribute grammar paradigm is sufficiently powerful to support the kind of attribution needed for configuration management. In particular, it is unclear whether there is a convenient usable mechanism for supporting tree/graph transforms. Supporting dynamic systems suggests the need for the extension and attribution transformations to be intertwined. The problem is that the result of an attribution process may be a new extended graph that then needs to be attributed. While there has been some work to formalize the automatic tree transformation process[VSF89] the results are inconclusive. Other related work in the attribute grammar domain focuses on incremental attribution[sKK86,RTD88]. There has also been direct research into applying incremental attribution to the configuration management process[Pfr86,WF88].

The extension process is based on the notion of type inferencing. The problem is to find the appropriate compromise between the expressive power of the type system and the ability to infer legal phrases[CW85]. New work in type inferencing [Weg87] may aid in finding that compromise.

Object management plays a significant role in the configuration management process. It is hard to envision future systems in which there is no object manager. It is important to be aware of the state of the art in object management and if possible try to influence its evolution by proposing requirements. Relational schemes, object oriented schemes, entity-relationship models, deductive databases have all been suggested. Key issues such as distribution, version management[Zdo86], caching[HK86], syntax and semantics are still not fully understood. In one sense the evolution of configuration management in particular, and SDEs in general, depend on real progress in object oriented databases. Conversely, requirements defined though progress in environment research will drive (direct) progress in the database domain.

Consistency constraints are an integral component of the model described above. Some analysis into the complexity of maintaining a set of constraints is discussed in the SIO system[GLL88].

Another key related issue is the underlying operating system that provides support for the configuration management process. With a trend towards more powerful workstations the notion of distributed configuration management is becoming more and more important. While one can argue that some of the issues concerning distribution can be pushed up to higher levels, it seems that the more the operating system can support the less there is to be coded into each application. Two systems that exemplify desirable qualities in future operating systems are Mach[8] and CPR[CM87]. Mach is an operating system developed to support distributed processing via the notion of light-weight processes. CPR, a research prototype, is a system which supports the notion of "database memory". In CPR the system supplies and supports physical locking and journaling for certain segments. The application programmer writing a database system, for example, need only worry about the semantics of the database (how to implement join, etc), not what and when to lock.

Configuration management relies on support from programming languages to provide information to simplify the process. It also relies on the database community to provide a mechanism for persistent, efficient storage of, and access to, this information. In some systems there is even builtin support for configuration management within the host operating system[Leb84].

Languages, and language design play a significant role in configuration management. Language design is evolving to meet the needs of programming-in-the-large. Modular languages, separation of interface from specification and polymorphism are some of the results of this evolution. Languages such as Modula-2, ADA, and CLU are examples of languages that provide some support for programming-in-the-large. Configuration management systems are using this extra information to help the CM process. For example, the separation of interface from specification in Modula-2 has allowed for the creation of CM systems that perform inter-module semantic analysis. In the future, it is quite possible that configuration management systems will be based more and more on information provided by compilers. Languages will either be modified or created in order to supply the appropriate information.

5 Future Work

The work outlined in this proposal represents a significant step towards the creation of a comprehensive model. Most details for the process of configuration construction are well understood and represented in the model. The details of configuration execution, on the other hand, are outlined but not fully integrated.

The focus for the remaining work centers around building an executable prototype of the model. Producing a prototype serves two important goals: it validates the model and forces details currently not specified to be filled in. In the process of creating the prototype the model will undergo changes. For the most part these changes will reflect refinements and modifications influenced by the validation process. Some of the model's details that need refinement are discussed below:

- **Attribution:** In the paper attribute grammar style attribution is proposed. One of the goals of future research is to determine if the kind of attribution needed to support configuration can be formulated entirely within this paradigm. The belief is that for most cases attribute grammars should be sufficient.

- **Extension:** The extension process proposed depends on type inferencing. Initially the the type system will be quite strict, and the inferencing a simple form of transitive closure. An area of research is to determine what impact a less strict type system (including tool-types and versions of types) will have on the model.

- **Tree Transformations:** One of the most difficult problems facing the model is an efficient solution to the tree transformation problem. The model supports dynamic attribution and extension. This implies the intertwining of attribution and extension processes. While this can be programmed-out in a straight forward manner, the issue is whether or not there is a formal mechanism capable of describing these transformations, that is consistent with the proposed attribution and extension transformations.

- **Homomorphisms:** The exact details of the transformations involved in creating the Product Configuration Graph from the Component Product Graph need to be refined. In particular, the details of the *is-component-of* relationship between the primitive objects and the derived-objects needs clarification.

- **Optimizations:** How the optimizations integrate with the model is the significant area of future research. It will be difficult to explore other optimization techniques until the mechanisms for timestamps and value-equivalence are fully integrated and operational.

The model proto-type will be implemented in Prolog. Prolog is a suitable implementation language because it allows for rapid prototyping and provides a primitive deductive knowledge base. As the implementation evolves it is expected that the model will evolve as well. Difficulties in implementation may lead to modification of the model. Issues not yet conceived may surface once ideas are explored using the model. Simplifications may even be possible based on future insight. Roughly, the areas to be covered in order to satisfy the thesis goals are:

- Refine the extension process in Product Configuration Graph phase.
- Clarify graph reduction and scheduling.
- Implement an executable prototype.
- Refine the model based on future work.

References

[8] ?. Mach:. In *Proceedings of the USENIX Conference*, Winter 1988 ?

[AH87] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *OOPSLA*, pages 420–440, October 1987.

[Baa88] Eric H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, Spring 1988.

[Ber87] Philip Bernstein. Database system support for software engineering. In *Proceedings 9th International Conference on Software Engineering*, Monterey, California, March 1987.

[BL85] Th. Brandes and C. Lewerentz. *GRAS: a Non-standard Data Base System within a Software Development Environment.* Volume 186 of *Lecture notes in Computer Science*, Springer-Verlag, New York, 1985.

[BLLV87] Y. Bernard, M. Lacroix, P. Lavency, and M. Vanhoedenaghe. Configuration management in an open environment. In *1st European Software Engineering Conference*, Strasbourg, France, September 1987.

[Bor86] Ellen Borison. A model of software manufacture. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[Cle86] G.M Clemm. *The Odin System - An Object Manager for Software Environments.* PhD thesis, Department of Computer Science, University of Colorado, Boulder, Colorado, 1986.

[Cle88a] Geoff Clemm. The odin specification language. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[Cle88b] Geoff Clemm. Personal communications. November 1988.

[Cle88c] Geoff Clemm. The workshop system - a practical knowledge-based software environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston Massachusetts, November 1988.

[CM87] Al Chang and Mark Mergen. 801 storage: architecture and programming. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, Austin, Texas, November 1987.

[CW85] L. Cardelli and Peter Wegner. On understanding types, abstraction and polymorphism. *Computing Surveys*, 471–521, December 1985.

[DL86] Klaus Dittrich and Peter Lockmann. Damokles - a database syetem for software engineering environments. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[Est85] J. 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.

[Fel79] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software - Practice and Experience*, 9(4):255–265, April 1979.

[Fel88] Stuart I. Feldman. Evolution of make. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[Fow85] G. Fowler. The fourth generation make. In *Proceedings of the Summer USENIX Conference*, Summer 1985.

[GGJ78] M. R. Garey, R. L. Graham, and D. S. Johnson. Performance guarantees for scheduling algorithms. *Operations Research*, 26(1):3–21, 1978.

[GLL88] E. P. Gribomont, M. Lacroix, and P. Lavency. Consistency if compatablity constraints in configuration management. *IEEE Transactions on Software Engineering*, 1988.

[Hen88] Peter Henderson, editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM, Boston Massachusetts, November 1988.

[HKM87] R. Hood, K. Kennedy, and H. A. Muller. Efficient recompilation of module interfaces in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Devlopment Environmemts*, January 1987.

[HS88] Svein O. Hallsteinsen and Ole Solberg. *Configuration Management - Concepts and Parctice.* Technical Report, University of Trondheim, Trondheim, Norway, 1988.

[HSO87] Dennis Heimbigner, Stanley Sutton, and Lee Osterweil. *APPL/A : A Language for Managing Relations Among Software Objects and Processes.* Technical Report, University of Colorado, Boulder, Colorado, September 1987.

[KC87] R.H. Katz and E. Chang. Managing change in a computer-aided design database. In *Proceedings of the 13th VLDB Conference*, Brighton, England, 1987.

[KHZ82] Uwe Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator.* Volume 141 of *Lecture notes in Computer Science*, Springer-Verlag, Heidelberg, 1982.

[LCS88] 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.

[Leb84] Leblang. Computer aided software enginerring in a distributed workstation environment. In *SIGPLAN notices*, May 1984.

[M75] J. Rochkind M. The source code control system. *IEEE Transactions on Software Engineering*, 1:364–370, 1975.

[MACK86] Terrence Miller, Jim Ambras, Martin Cagan, and Nancy Kendzierski. The unified programming envirionment: unobtrusive support. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[MK88] 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, Febuary 1988.

[MW87] K. Marzullo and Douglas Wiebe. Jasmine: a software system modeling system. In *SIGPLAN Notices*, pages 121–130, January 1987.

[Nar88] K. Narayanaswamy. Version control in the common lisp framework. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[Nes86] John Nestor. Toward a persistent object base. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[Ost86] Leon Osterweil. A process-object centered view of software environment architecture. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.

[Pen79] Maria H. Penedo. The use of a module interconnection language in the sara system design methodology. In *Proceedings 4th International Conference on Software Engineering*, pages 294–307, September 1979.

[Pen87] D. Jason Penny. Class modification in the gemstone object-oriented dbms. In *OOPSLA*, pages 111–117, October 1987.

[Per87] D. Perry. Software interconection models. In *Proceedings 9th International Conference on Software Engineering*, pages 61–69, March 1987.

[Pfr86] Mary Patric Pfreundschuh. *A Model for Building Modular Systems Based on Atribute Grammars.* PhD thesis, Department of Computer Science, University of Iowa, Iowa City, Iowa, 1986.

[Rai80] K. Raiha. Bibliography on attribute grammars. In *SIGPLAN notices*, March 1980.

[RTD88] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis ofr language based editors. *ACM Transactions on Programing Languages and Systems*, 1988.

[SCB86] Craig Schaffert, Topher Cooper, and Bruce Bullis. An introduction to trellis/owl. In *OOPSLA*, pages 9–16, September 1986.

[sKK86] s. Kaplan and Gail Kaiser. Incremental attribute evaluation in distributed lanuage-based environments. In *Proceedings of the ACM SIGACT/SIGOPTS Symposium on Principles of Distributed Computing*, Calgary, Alberta, Canada, August 1986.

[SZ87] Karen E. Smith and Stanley B. Zdonik. Intermedia: a case study of the differences between relational and object-oriented database systems. In *OOPSLA*, pages 452–465, October 1987.

[Tic82] Walter Tichy. Design, implimentation and evaluation of a revision control system. In *Proceedings 6th International Conference on Software Engineering.*, Tokyo, Japan, September 1982.

[Tic86] Walter Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[Tic88] W. Tichy. Tools for software configuration management. In *Proceedings International Workshop on Software Version and Configuration Control*, Grassau, West Germany, January 1988.

[TO88] Richard Taylor and Leon Osterweil. Foundations for the arcadia environment architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Boston Massachusetts, November 1988.

[VSF89] H. Vogt, S. Swierstra, and M. F.Kuiper. *Higher Order Attribute Grammars (Draft).* Technical Report, University of Utrecht, Utrecht, The Netherlands, 1989.

[Weg87] Peter Wegener. *The Object Oriented Claaification Paradigm. Computer Science Series*, MIT Press, Cambridge Massachussets, 1987.

[WF88] Mary Pfreundschuh Wagner and Ray Ford. Using attribute grammars to control incremental, concurrent builds of modular systems. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[WHK88] William Waite, Vincent Heuring, and Uwe Kastens. Configuration control in compiler construction. In *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[Wie86] Douglas Wiebe. A distributed repository for immutable persistent objects. In *OOPSLA*, pages 453–465, September 1986.

[Wie88] D. Wiebe. Specifying and verifying semantic properties of software configurations. In *Proceedings International Workshop on Software Version and Configuration Control*, Grassau, West Germany, January 1988.

[Win88] Jurgen F. H. Winkler, editor. *International Workshop on Software Version and Configuration Control*, Tubner-Verlag, Grassau, West Germany, January 1988.

[Zdo86] S. Zdonik. Version management in an object-oriented database. In *IFIP WG2.4 International workshop on Advanced Programming Environments*, Trodheim, Norway, June 1986.