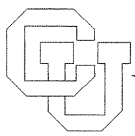
Towards a Graph Transform Model For Configuration Management Environments

Dennis Heimbigner and Steven Krane

CU-CS-383-88 January 1988



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in corporation with the Defense Advanced Research Project Agency and the IBM Corporation

Towards a Graph Transform Model for Configuration Management Environments
(An Extended Abstract)

Steven Krane, IBM T. J. Watson Research Center
Dennis Heimbigner, University of Colorado

To appear in Proceedings of the International Workshop on Software Version and Configuration Control, Grassau, FRG 27-29 Jan. 1988

A concise, simple model for software configuration management that subsumes many existing systems is described. The model is based upon transactions and the transformation of graphs to describe the steps in configuration management.

---

1. Introduction

Software configuration management is the process of controlling the changes to a collection of software objects. A number of software configuration management systems have been proposed and a smaller number have been implemented effectively (e.g., [1, 3-5]). Despite the amount of research in this area, there seems to be little agreed-upon terminology and even less understanding of how the various existing configuration managers relate to each other.

The goal of this paper is to define a concise and consistent model for software configuration management environments that subsumes a large number of configuration managers (in particular Odin [1], Adele [3], Make [4], and DSEE [5]). Given such a general model, it is possible to compare and contrast systems using their representation within this general framework. It is also possible to define and implement the general model to explore its utility as a replacement for these other configuration management systems.

The rest of this paper is devoted to a description of the model. It is necessarily brief; in particular, it does not consider issues of execution strategies or issues of implementation. It also assumes that the reader has some familiarity with the configuration managers listed in the references.

2. Modelling Approach

Our approach to modelling explicitly recognizes the transactional nature of configuration management. That is, such systems are driven by requests for software objects. This explicit transaction orientation is one feature that differentiates our model from most others [1].

In the model, an object request is an (incomplete) specification of a specific software object or objects (sometimes termed the *product*). This specification of an object also includes the specification of the attributes of that object such as "compilation_level = OPTIMIZE". The process for obtaining a product consists of: (1) repeatedly transforming an object specification to produce another, more detailed, specification and (2) executing that

[1]Most systems we have encountered implicitly are transactional in nature, but there is virtually no mention of this in the literature.

specification.

All specifications in this model take the form of attributed graphs. The nodes of the graph are software objects. Each type of software object has an associated set of attributes. The edges represent various relationships over the objects. In the model described subsequently, the relationships are *is-component-of* and *derives*. The *is-component-of* relation represents the use of one component by another component. We define *component* to refer to a non-re-derivable object. The *derives* relation represents the transformation of one object into another object. Both relations may be viewed as special cases of a more general *depends-on* relation.

Two kinds of generic transformation processes are used in this model: *extension* and *attribution*. An extension transformation takes a graph and adds nodes and edges corresponding to some particular relationship. This transformation actually changes the shape of the graph. An attribution transformation propagates the values of attributes to any nodes that do not have the attribute values defined. As we shall see, the model operates by alternating extension and attribution transformations. The process continues until all relations have been extended and all attributes propagated.

2.1. Knowledge Base

The transformation processes described here utilize an underlying knowledge base containing the following information: (1) All software objects, (2) Transformation rules (both for extension and attribution), (3) Relationships between objects, and (4) the attribute values of objects.

The *depends-on* relation is a principal kind of information contained in the knowledge base. In our model this information is conveniently represented in two graphs: the *Uses Graph* and the *Type Graph*. They appear in one form or another in almost all configuration managers. As shown below, the way in which these graphs are used distinguishes one system from another.

The Uses Graph (UG) defines transformation rules for specific software objects. It may specify both extension and attribution rules. The nodes of this graph are software objects and the edges are instances of the the *derives* relation. The Make system uses a Makefile, which is mostly a representation of the UG. For example, a Makefile might contain the following dependency line:

```
test.o:   test.c test.h
```

This line represents the specific dependency information for the specific object `test.o`.

The Type Graph (TG) also contains transformation rules. However, these rules do not involve specific objects; rather they are generic rules that are defined in terms of object *types*. The nodes in this graph are the types and the edges represent the *derives* relationship. These edges can be used to extend a graph along the *derives* relation. In the Make system, this graph is represented by the so-called default rules such as this:

```
.c.o:
        <commands to convert a .c object to a .o object>
```

Associated with the TG is a set of graphs isomorphic to the TG: they have the same shape as the TG but carry different semantic information. Each graph (termed an *Attribute Graph* (AG)) in this set defines the propagation rules for some specific attribute. Some of the attributes that might be covered in the set of AG's are: *tool*, *debug*, and *name*.

2.2. Model Overview

In the following sections we describe the steps in the model by characterizing the graphs they produce. In all cases these graphs are the result of applying a transformation process to a previous graph (except for the initial transaction request). Where possible, reference is made to the existence of specification languages for describing these transformation processes. Finally, we refer back to the system that drove the inclusion of the graph in the model. See Appendix I for a pictorial representation of the model.

2.2.1. Partial System Graph

The *Partial System Graph* (PSG) represents an initial, very high level, specification of the object(s) (i.e., product) to be produced. In practice, this graph usually consists of a single root (object). In Adele, the PSG is represented by the *configuration*. In Make it is the *label* used to describe what to make (e.g. *make compiler*). In DSEE it is the root of the loaded system model. In Odin it is the combination of the type of the product plus some source object. Typically the PSG is selected by a request from the user (*make compiler, build, etc*).

2.2.2. System Graph

The *System Graph* (SG) represents the list of non-re-derivable elements needed to build the product. This is sometimes called a *composition list*[2]. The SG is derived by starting with the PSG and extending that graph via the *is-component-of* relationship. In most systems, this relationship is statically defined in the knowledge base. Thus the extension process consist of the trivial operation of attaching that static information to the PSG.

Adele is an example of a system in which this process is dynamic. Since the Adele system supports Modula-2, it separates our notion of a component into Modula-2's interface plus a body. An Adele *Configuration* is a high level specification of the process for building the composition list. It is a rule based language that describes attribute selection and constraints on those attributes. The attributes generally describe the various dimensions of multi-version properties. For example, the rule "system = UNIX and machine = MC68000" directs the selection of particular bodies of the needed interfaces. It is important to note that this selection of specific bodies then drives the selection of further interfaces, which in turn drive the selection of more bodies. For example it may be that the MC68000 body of the "device_driver" interface uses the "m68_device_generic" interface while no other body for "device_driver" does. Given the Adele Configuration (PSG) and the state of the knowledge base it is quite possible to end up with different composition lists.

2.2.3. Bound System Graph

The Bound System Graph (BSG) represents the result of binding a fully qualified name to the components needed to construct the product. The BSG is constructed from the SG by setting the values of the name-related attributes. The rules for specifying the values are taken from the knowledge base and depend on the particular system being modelled.

The names as represented in the SG typically represent the base names of the needed components. These names do not include information about version identification, revision identification within that version, or any hint as to the physical location of the object. The process that constructs the BSG assigns appropriate values to the version, revision, location, and other attributes associated with the nodes of the BSG graph.

In Make, this process does not exist. The names given in the makefile **are** the fully qualified names. In DSEE, on the other hand, the *Configuration Thread* (CT) provides a

rather elegant mechanism for the high level specification needed to complete the version and revision attributes. An English paraphrase of a typical CT could read as "*Use all of the files that I have checked out plus release 5 of foo.bar and release 4 of everything else.*"

2.2.4. System Object Graph

The *System Object Graph* (SOG) describes the complete set of objects needed for the construction of the product. In this context, complete means that both the non-re-derivable objects mentioned above and those associated re-derivable objects are part of the final product. This graph is constructed by extending the BSG via the *derives* relationship. This transformation adds both new software objects as nodes and new edges of the *derives* relationship. For most systems there is no language to specify the process for building the SOG. One obvious area of future research is to explore what constructs are needed to specify this process.

In most systems, the dependency information, as described in the UG and TG, specifies the transformation rules for extending the *derives* relation. We distinguish the generic rules found in the TG from the specific rules found in the UG. In Odin the TG is used exclusively; there is no notion of the UG. The addition of edges and nodes to the BSG is inferred solely from the type structure described in the TG. Systems other than Odin rely heavily, if not exclusively, on the UG. In Make, for example, `test.o` may be taken to specify a rule for extending the BSG by connecting together the nodes `test.o`, `test.c`, and `test.h`. Make can also use default rules (its equivalent of the TG) to infer the addition of new edges and nodes to the BSG. Make can only perform single step inferencing, while Odin can perform multiple-step inferencing. Thus, in this sense Odin has a more powerful inferencing mechanism than Make. Adele and DSEE, by contrast, do not have any form of inferencing at all.

2.2.5. System Configuration Graph

The *System Configuration Graph* (SCG) is the last graph in the transformation sequence (at least when modelling [1, 3-5]). All the information needed to construct the final product is explicit in this graph. This process is also an attribution process. It takes the SOG and sets the values of some attributes for some objects in the BSG. Some of these values are taken from the original transaction specification. Others are implicitly added by the system (e.g., the tool attribute). The next step propagates the attribute values to all nodes of the graph. The rules that drive the propagation are specified in the knowledge base. For example, if the user requested that the product's debug attribute value be *true*, then this value would be set accordingly in the PSG. It is here, in the more detailed graph that the value for the debug attribute is propagated to all the appropriate objects.

Odin makes use of this process more so than the others. This is where the Attribute Graphs described earlier are most often used. For the above example, both the *tool* and *debug* AG's would be used to finish decorating the graph.

3. Future work

The basic structure of this model has been defined and our next step will be to formalize it and to use it to model the details of the other configuration management systems mentioned previously. This will require us to define the exact set of rules (both general and specific) to be used in the graph transformations such that the effect is the same as the system being modelled.

We believe we understand how to model the transformations embodied in the systems referenced in this paper. But much work remains to be done to capture the execution semantics. Some of the issues to be addressed in the execution process are:

(1) Change propagation: Most systems decide when to re-construct an object based on time stamps. However, there are other possibilities such as direct comparison of old and new values, and tool based equivalence tests.

(2) Tool Scheduling: Given a specification, it may be possible to execute portions of the specification in parallel (for example, to speed up the execution time).

(3) Evaluation Strategies: Most of the systems described use a lazy evaluation scheme. Other evaluation schemes exist (eager, opportunistic, etc). To what degree do these other strategies impact the model?

Finally, we intend to construct a working software configuration manager based on this model in order to evaluate its utility. One aspect of this research will be to analyze techniques to optimize the overall process.

4. Acknowledgements

The authors gratefully acknowledge the support of the National Science Foundation grant #DCR-8745444 in cooperation with the Defense Advanced Research Project Agency, and the IBM Corporation.

5. References

1. G. M. Clemm, *The Odin System - An Object Manager for Software Environments*, PhD Thesis, Dpt of Computer Science, University of Colorado, Boulder CO , 1986.

2. E. Cristofor, T. A. Wendt and B. C. Wonsiewicz, ''Source Controls + Tools = Stable Systems'', *Proceedings 4th Computer Science and Applications Conference*, October 1984, 527-532.

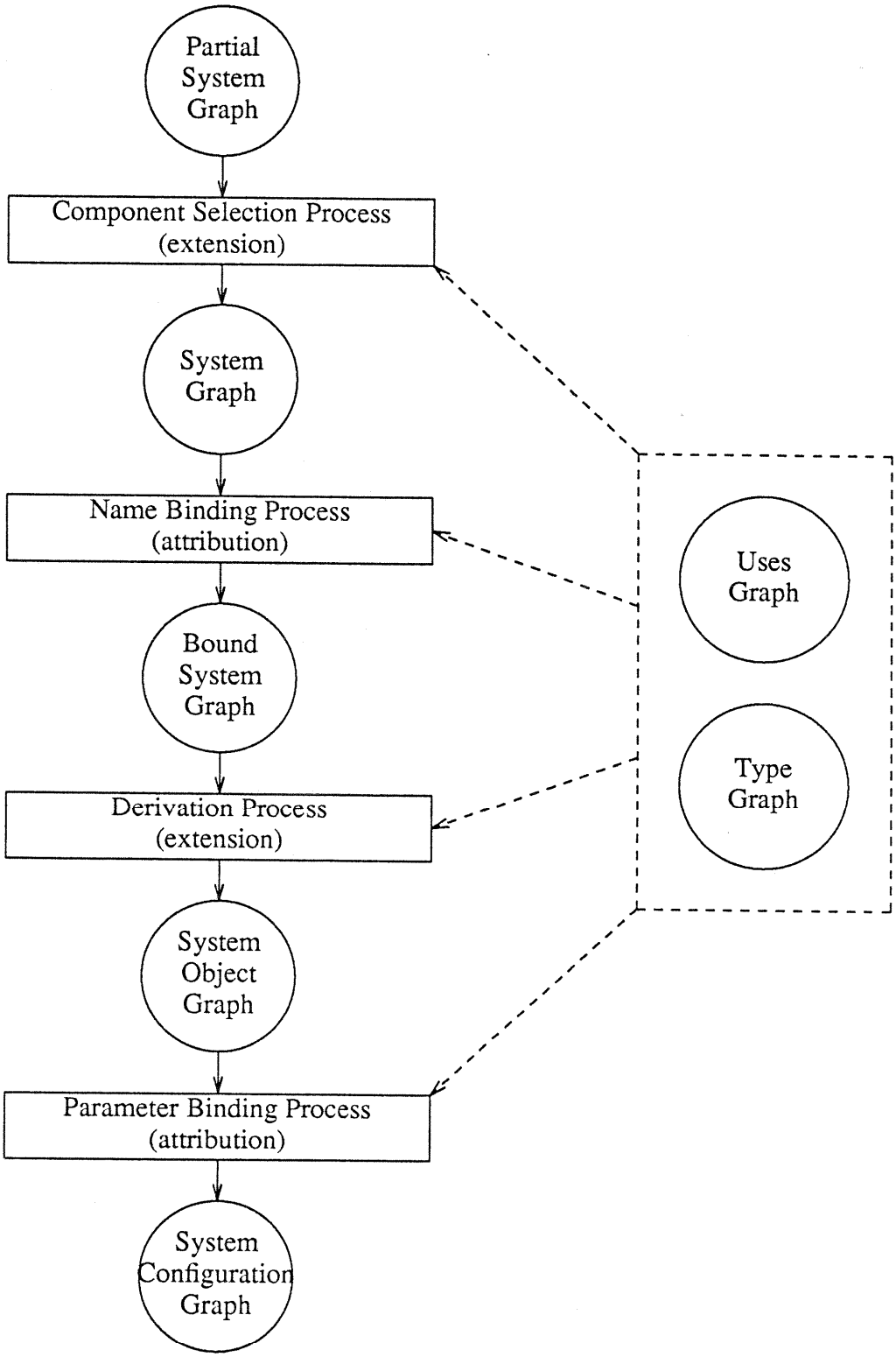
3. J. Estublier, ''A Configuration Manager: The ADELE Data Base of Programs'', *Workshop on Software Engineering Environments for programming-in-the-large*, Harwichport, Mass, June 1985, 140-147.

4. S. I. Feldman, ''Make - A Program for Maintaining Computer Programs'', *Software - Practice and Experience 9*, 4 (April 1979), 255-265.

5. D. B. Leblang, R. Chase and G. M. McLean, ''The Domain Software engineering environment for large scale software development'', *IEEE Conference on Workstations*, November 1985.

Steven Krane (krane@boulder.colorado.edu)
Dennis Heimbigner (dennis@boulder.colorado.edu)
Computer Science Department
University of Colorado
Boulder, CO 80309-0430
USA

Appendix I



Pictorial Representation of the Graph Transform Model