The Data Model of the Configuration Management Assistant

Erhard Ploedereder, Ph.D.
Adel Fergany, Ph.D.

Tartan Laboratories Inc.
300 Oxford Drive
Monroeville, PA 15146

Abstract

In an environment in which systems are configured by reusing existing subsystems, the determination of complete and consistent configurations is a non-trivial and error-prone task, although considerable information about the subsystems may already be available from previous configurations. The Configuration Management Assistant is a tool that supports tracking and exploiting such information in the difficult process of re-configuration on a large scale. Its data model was designed to be as independent as possible of configuration management policies and procedures and yet provide substantive assistance in this process. The most important elements of this data model are described in this paper.[1]

1. *INTRODUCTION*

Configuration Management is a central activity in the software generation process. Today, many projects flounder or are considerably delayed by problems that originate from insufficient discipline or technical support for configuration management. Progress in the area of configuration management is desperately needed.

Configuration management is often mistakenly considered to be satisfied by a capability to track changes to program source code [8, 7] and to automatically regenerate a system after such changes have taken place [4]. While these activities play an important role in software management and change control, they are nevertheless not the primary, let alone the sole focus of configuration management. For such capabilities, the GIGO (''garbage-in-garbage-out'') law applies: if the programmers are not capable of determining a functionally consistent set of sources to begin with, no degree of automation will make the resulting system operational. Moreover, the ease with which systems can be rebuilt automatically in this fashion makes it even more likely that functional inconsistencies are not detected at an early stage of system integration, but only after considerable time and effort has been wasted on building and testing a non-functioning system.

These problems become even more pronounced in an environment in which sizeable software components are heavily reused. Here, the internal dependencies within the reused software are largely unknown to the (re)user, but may have subtle interactions with other software that need to be accommodated in the system integration. Conversely, as software gets reused in different contexts, a wealth of information is accumulated about consistent and inconsistent compositions. Utilizing this information can significantly increase the probability of arriving at consistent and complete configurations even for newly created configurations, in which major subsystems have been replaced. Tracking and exploiting such information and making it available to the (re)user is the central objective of the Configuration Management Assistant (CMA). While many of the basic concepts employed in CMA are also found in other systems, it is the integration of such concepts as logical dependency, consistency, compatibility, abstract decomposition, and configuration semantics that distinguishes CMA.

[1]Work reported herein was performed by Tartan Laboratories, 300 Oxford Drive, Monroeville, PA 15146, for the Naval Avionics Center, Indianapolis, under contract no. N00163-98-C-0148.

2. *THE CONFIGURATION MANAGEMENT ASSISTANT (CMA)*

The Configuration Management Assistant (CMA) is a tool that assists in configuration management by

- recording and retrieving descriptions of configurations and the set of entities of which they are comprised;
- recording and retrieving information about known (in)consistencies and dependencies between individual entities that are included in configurations;
- predicting the completeness and consistency of newly created configurations based on the recorded information, and assisting in the construction of such new configurations.

The CMA is a powerful administration system for configuration management information and imposes as few constraints as possible on the specific policies and methods of configuration management that a particular CMA installation might wish to apply. The CMA offers open interfaces that can be used by other tools both to record and to retrieve configuration management related information. Enforcing particular management policies and procedures, as necessary to perform efficient configuration management, is left to other tools interfacing with the CMA and to locally established guidelines and procedures. Producing a CMA that supports a large variety of configuration management procedures and of styles to describe configurations has been a guiding principle and a major challenge in the design of the CMA.

The major advantage of using the CMA comes from its capability of using recorded information about dependencies and consistencies between entities to assist the user in defining workable configurations. The heavy expense of finding configuration errors through laborious debugging, once the attempt to build a product has failed, can be avoided. In providing these services, the CMA relies heavily on the wealth and correctness of the dependency and consistency information supplied by the user or by tools. Its capabilities are likely to be particularly strong in a production environment where components and tools are heavily reused across products, since earlier usages can provide such needed information reliably.

Source control systems, *e.g.* RCS [7], by which changes to individual entities are controlled and differences are tracked, can be integrated with the CMA through the use of intermediary tools or the recording of source control labels in the CMA data base. Similarly, tools for the automatic (re)generation of products, *e.g.* Make [4], can obtain a large portion of the information needed for such (re)generation from the CMA.

The CMA has been implemented by Tartan Laboratories Inc. under contract for the Naval Avionics Center, Indianapolis. It consists of the implementation of the Public CMA Access Interface in terms of Ada packages, on which diverse tools can be built, and of the interactive, menu-driven CMA User Interface that guides the user in building new configurations and in entering other information not already communicated by tools via the Public CMA Access Interface. The CMA is implemented in Ada; in March 1989 it consisted of about 45,000 lines of Ada code.

3. *BASIC CONCEPTS*

The CMA has the concepts of *logical objects* and their *instances*. A logical object may have multiple instances, distinguished by *key attributes* that characterize each instance. For example, different versions of an object will typically be instances of the same logical object. However, the CMA has no built-in knowledge of a specific scheme or policy of versioning. The CMA does not explicitly support a tree-structure of instances (*i.e.*, versions of versions). Instead, multiple key attributes can be used to reflect subordinated versioning; nevertheless, all versions are regarded as direct instances of their logical object. This relieves the user of the necessity of an a-priori decision regarding such tree structures and avoids the potential need for entity duplication to conform to a tree-structured division of information.

The CMA has the concept of *composite logical objects*, consisting of a set of other (possibly composite) logical objects. Composite logical objects are used to describe the abstract decomposition of large systems into their subsystems, called *components*. A component may belong to multiple composite objects and, in particular, may be a component of multiple subsystems of a single composite logical object. In other words, system decomposition need not be tree-structured, but can be represented by an acyclic graph of subcomponents. For example, the logical object "compiler" may be composed of a "front end" and a "back end" logical object, both of which in turn have substructure containing the same logical object "error writer" as component.

The CMA has no built-in knowledge of the nature of the

A *rendition attribute* is predefined to exist on each instance. The attribute and its user-defined values are intended as a means to distinguish the nature of the instances, to delineate whether or not multiple instances of the same logical object can occur in a configuration, and to describe, by implication, the nature of a configuration.[2] Examples of rendition attribute values might be "Source", "Object", "Ada Library", "Documentation", "Specification", "Body", etc. Similar to partition attributes values, the values in the value set of the rendition attribute may be specified as being specializations or generalizations of other such values. The following specific semantics apply (see also Section 5):

1. a dependency, constraining the rendition of any satisfying instance, can be satisfied only by any instance whose rendition generalizes the given rendition.
2. an inheritable dependency, constraining the rendition of the source instance of inherited dependencies, is applicable only to instances whose rendition generalizes the given rendition.
3. a compatibility relationship can exist only, if the rendition of the target instance generalizes the rendition of the source instance.
4. a configuration cannot include two or more (used) instances of the same logical entity and of overlapping rendition.

By applying specialization and generalization semantics to rendition attribute values, CMA is capable of tracking dependencies and other relationships among administered objects even if their aggregation changes in different versions. For example, at early stages of a development, a given instance might contain both specification and implementation of a module while, later on, specification and implementation are separated into different instances. CMA nevertheless will be capable of creating configurations including either representation of the module, based on the same dependency information.

Version attributes are used to distinguish variations of instances of the same partition and rendition, *e.g.*, "V1" and "V2" of a library, "debug" and "optimized" of a module, etc. There is no specific CMA semantics associated with these attribute values (other than the overall requirement of unique identification of instances by attribute values). Each instance may have multiple version attributes.

Within configurations, instances have a *purpose attribution*, distinguishing whether the instance is produced by some production step utilizing the configuration description, or used in the production of another instance, or both.

5. *THE RELATIONSHIPS IN THE CMA DATA MODEL*

Component relationships exist among logical objects. Such a relationship specifies that a given logical object is a component of a composite logical object. Each logical object can have multiple logical objects as components as well as be a component of multiple composite logical objects. These relationships are used to describe the composition of logical objects in a "top-down" fashion. The primary purpose of these relationships is to create a system decomposition that can serve as the description of a configuration family in terms of its logical objects. Each component relationship can be attributed with a rendition attribute value to indicate that the relationship is valid only for configurations that are not constrained to exclude instances of a rendition that generalizes this given rendition (see Section 6). The system composition can also be described in a "bottom-up" fashion by means of dependency relationships discussed below.

Instance relationships relate logical objects and their instances. While every logical object may have multiple instances, no such instance can belong to multiple logical objects.

Figure 1 depicts the component and instance relationships of the example of a compiler that has two components, front end (FE) and back end (BE), both of which require an error writer (EW). This figure shows several specification ("spec") and implementation ("impl") instances of FE, BE, and EW. It is assumed that the user has defined "spec" and "impl" as rendition attribute values, and "VMS" and "Unix" as partition attribute values; both "V1" and "V2" are version attribute values.

Logical dependency relationships exist between instances and logical objects. They specify that the given instance depends on, *i.e.*, cannot operate or be processed correctly without, or depends in its correctness on, some instance of the given logical object. Any instance may

[2] The concept of rendition is very close to the concept of typing. As the rendition semantics are different from Ada typing, we consciously avoided the use of the word "typing".

entities it administers. It merely expects the user to provide strings presumed to identify the entities. These strings could be (the names of) host files, directories, Ada libraries, labels used by a source control system to identify appropriate host entities, command scripts to generate the entities, or anything the user chooses them to mean. These strings could differ in their user-defined semantics for different kinds of administered entities. Instances of a logical object can be of quite diverse nature: A logical object representing an Ada package may have as its instances the package specification and the package body in source form, assembly or object files compiled from these sources, separate documentation, *etc*. Also, each of these instances may exist in different variations and revisions. Similarly, a logical object representing a tool may have as its instances the tool's executable image, its user documentation, its internal documentation, or the configuration used to build the tool. Consequently, the CMA can be applied to many diverse configuration management problems. The CMA can be viewed as an elaborate mechanism for name storage and retrieval, highly structured and supported by the knowledge the CMA has about configurations that interrelate the administered entities. Administered entities need not even exist at the time they are made known to the CMA; it may, in fact, be the very purpose of the configuration to control the generation of these entities.

The CMA concept of a *configuration* is simply that of a set of instances. Based on information recorded about these instances, completeness, consistency, and satisfaction of uniqueness constraints of the configuration are decided. The CMA also has the concept of a *configuration family*, describing a not fully determined set of instances in terms of constraints imposed on the eventual selection of instances for a particular configuration.

The data model used by the CMA is an ERA (entity-relationship-attribute) model, in which both entities and relationships can be attributed. Logical objects, instances, configurations and configuration families are such entities. The CMA implementation supports most relationships only as directional relationships.

4. *THE ATTRIBUTES IN THE CMA DATA MODEL*

Instances are characterized by (user-defined) attributes. These attributes are presumed to be detailed enough to allow the user the selection of the instance of a logical object, matching the purpose of the configuration to be built. The CMA requires that the set of attributes and their respective values on instances of a logical object be sufficiently distinct to uniquely identify the respective instances. New attributes can be declared at any time and henceforth used on instances to be created. The value sets of attributes can be extended at any time by the user. The semantics of the attributes depends on their respective kind. There are three different kinds of key attributes: partition, rendition, and version attributes. The kind of an attribute is determined when it is declared to the CMA.

Partition attributes are intended as a means to distinguish those instances of all logical objects in a configuration family that are guaranteed to never be in the same configuration. For example, some modules may be host-dependent; in this case, a partition attribute named "host" and valued "VMS" or "UNIX" can be defined and used for the respective modules. The semantics conveyed to the CMA is that, for a configuration built for a "UNIX" host, modules with a "host" attribute must have the value "UNIX". Each instance may have multiple partition attributes.

In order to create nested sets of successively more detailed partitions, a partial order can be induced on partition attribute values: the values in the value set of a given partition attribute may be specified as being specializations or generalizations of other such values. Any value may have many specializations and generalizations. As an example of the use of specialization and generalization, instances can be partitioned according to a "target" partition attribute and its values "register-based architectures" with specializations "Intel-386" and "MC68k" with further specializations "MC68020", and "MC68030". Since "register-based architectures" generalizes both "MC68k" and "MC68020", a configuration can then be built from instances with the target partition attribute values "register-based architectures", "MC68k" or "MC68020" (or without a target partition attribute). However, no configuration can be formed that includes "Intel-386" and "MC68020" targeted modules as these two values are incompatible. Similarly, "MC68030" and "MC68020" are incompatible and may not co-exist in the same configuration.

By definition, for any two values A and B of the same attribute, A is said to *generalize* B, if and only if A is equal to B or A is a member of the transitive generalizations of B. Similarly, A *specializes* B if and only if B generalizes A. Finally, A *overlaps* B, if either of them is a transitive generalization of the other.

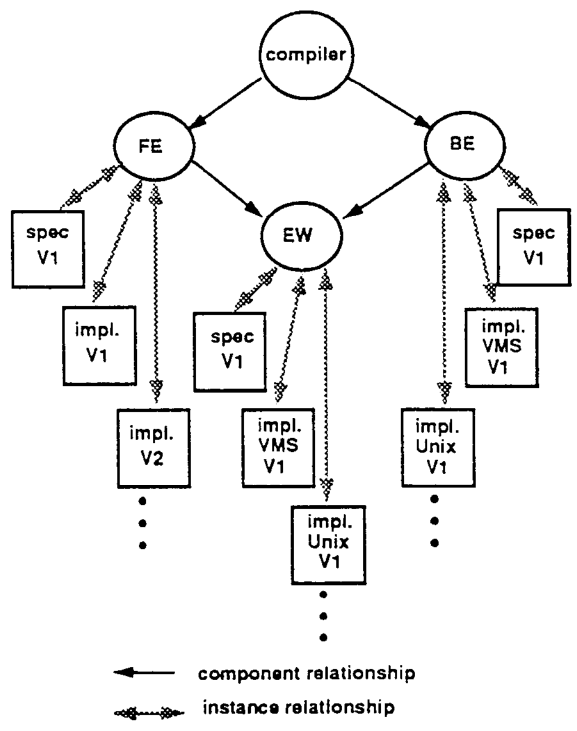


Figure 1: Component and Instance Relationships

have many such dependencies and a logical object can be depended upon by many instances. These relationships are attributed to distinguish between dependencies arising from using or producing the respective instance in a configuration; this is referred to as a *purpose attribution* of the respective dependency. They are also attributed with a rendition, indicating that, for an instance of the logical object to satisfy the logical dependency, its rendition must generalize the given rendition.

The logical dependency relationships are used in the process of establishing configurations. A *complete configuration* must include an instance of any logical object targeted by any applicable logical dependency emanating from any instance in the configuration. The rendition of the instance must generalize the rendition of the logical dependency. The same instance may satisfy multiple dependencies.

Inheritable dependency relationships exist from logical objects to other logical objects. They indicate to the CMA that, upon creation of an instance of a logical object from which such relationships emanate, logical dependency relationships are to be installed to the logical objects at which the inheritable dependency relationships terminate. Any logical object may have many such emanating and terminating inheritable dependency relationships. These relationships have a purpose attribution to distinguish between dependencies arising from using or producing the instance that inherits an equally attributed logical dependency relationship. They are also attributed with a *target rendition* to become the rendition of the inherited logical dependency relationship of an instance. In addition, they are attributed with a *source rendition* which limits the inheritability to those instances whose renditions generalize this source rendition.

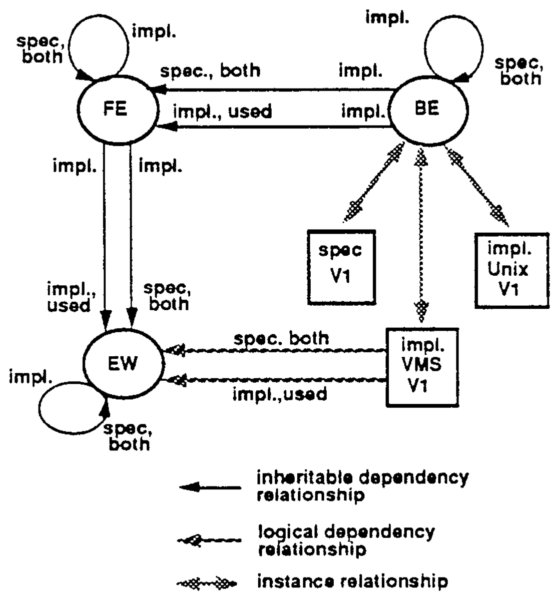


Figure 2: Inheritable and Logical Dependency Relationships

Figure 2 illustrates the inheritable and logical dependency of the compiler example. It is assumed that every implementation instance ("impl") requires a specification instance ("spec") of the same logical object, every front end (FE) requires an error writer (EW), and every back end (BE) requires a front end, but not every back end requires an error writer. Thus, all dependencies shown in Figure 2 are inheritable dependencies, except those of the "VMS" back end on the error writer which are logical dependencies. The purpose attributions specify that dependencies exist on specifications only, if the depending instance is to be produced (*i.e.*, compiled), or on both specifications and implementations, if the depending instance is to be used (*i.e.*, linked). This setup allows one to configure a stand-alone front end that includes an error writer, or a back end that includes a front end and an error writer. However, one cannot configure a back end that does not include a front end. Moreover, CMA ensures that, upon combination of a back end with a (previously configured) front end, only one version of an error writer is included in the resulting configuration.

Consistency relationships exist only between instances of different logical objects or of disjoint renditions. Any instance may have many consistency relationships. By boolean attribution of the relationships, known inconsistencies can also be recorded. These relationships specify that a logical dependency of a given instance can be satisfied by another instance which is (in)consistent with the former for any configuration that chooses the respective instances. What is meant by ''consistency'' is largely to be determined by the user of the CMA, although the intuitive intent is that of "correctly operating together"[3]. The CMA semantics of consistency merely requires that the user's notion of consistency satisfies three rules:

1. A pair of instances consistent in some configuration will be equally consistent in any other configuration that includes this pair of instances.
2. A configuration is to be regarded as a *consistent configuration* if, for every applicable logical dependency relationship emanating from an instance in the configuration, there either is a consistency relationship from the given instance to the instance selected for the logical object in this configuration and satisfying the dependency, or no such instance has been selected yet.
3. Consistency is independent of the purpose of inclusion of the respective instances in a configuration, *i.e.*, the consistency exists regardless of whether the depending instance is used or produced within the configuration.

Compatibility relationships are between instances of the same logical object and of certain related rendition. Any instance may have many compatibility relationships. These relationships specify a directional compatibility between the involved instances. Such compatibility is taken to mean that replacement of the source instance of the relationship by the target instance will preserve the consistency with any instance that depended on the replaced instance. Note that no such guarantee is required regarding any object depended upon by the compatible instances. That is, the replacing instance may well depend on logical objects and instances different from those of the replaced instance. The rendition of the source instance of a compatibility relationship must specialize the rendition of the target instance. Compatibility relationships are used to predict the consistency of a new configurations formed by (compatible) transformations of existing configurations. They are also used to guide the search for consistent instances that satisfy logical dependencies on some objects.

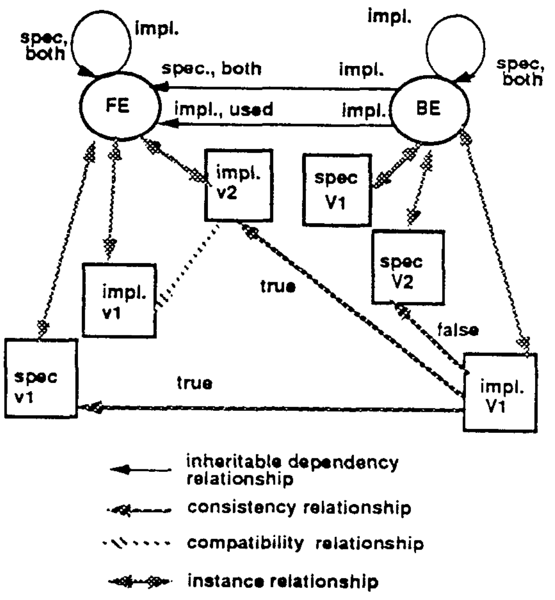


Figure 3: Consistency and Compatibility Relationships

Figure 3 illustrates the consistency and compatibility relationships on our example of a compiler. As a result of including the V1 implementation of BE as a used instance in a configuration, one must satisfy four inheritable dependencies. The dependency of this BE implementation on a specification of BE can be satisfied consistently only by including the V1 specification instance as the V1 implementation is known to be inconsistent with the V2 specification. The dependency on the FE implementation can be satisfied consistently either directly by the V2 implementation of known consistency, or by the V1 implementation by virtue of the recorded compatibility with the V2 implementation. The dependency on a FE specification can be satisfied consistently by inclusion of the V1 specification. The dependency of the FE implementation on an FE specification is already satisfied by the included V1

[3]Note that the CMA notion of consistency is not the absence of the conflicting inclusion of multiple versions of the same object in a configuration, as it is in systems like Gypsy [3]. CMA refers to the latter situation as a skewed configuration.

specification of FE. Note that, at this stage, the resulting configuration is still not regarded as a consistent one, since no consistency relationships are present that match the satisfied dependencies between the implementation instances and the specification instances of each logical object, *e.g.*, between the FE implementation and the FE specification instances. CMA allows the creation of configurations of such unknown consistency and even of configurations known to contain inconsistencies. Users can query CMA for display of the specific dependencies for which either consistency is unknown or an inconsistency has been detected. Furthermore, upon retrieval of any configuration, CMA will track any changes to relevant consistency relationships since the last retrieval and thus always present the most recent consistency information to the user of the configuration.

Configuration relationships exist between instances and those instances that contain configuration descriptions including the former instances. These relationships are implicitly maintained by the CMA and are attributed by the reasons for including instances in configurations. They serve to answer useful queries such as "what are the instances included in a given configuration?", "why is a given instance included in a given configuration, which instances depend on it and which ones does it depend on?", "which dependency caused a constraint on the instance selection?", "what are the configurations in which a given instance is included?".

Appendix A graphically depicts all the relationships among the entities of the CMA data model.

6. *CONFIGURATIONS*

The properties of *completeness* and *consistency* of a configuration have already been explained in the preceding section. Two additional properties of configuration descriptions are of interest:

A configuration description is said to be *ambiguous*, if it includes logical objects without selection of specific instances or logical objects depended upon without also including the appropriate instances to satisfy the dependencies. The choice of such instances may have already been narrowed by user-specified *selection constraints*. An ambiguous configuration can be consistent with respect to all the instances already included. It can be recorded and subsequently used to derive complete configurations.

A configuration description is said to be *skewed*, if it includes multiple used instances of the same logical object and overlapping rendition, thus violating the uniqueness restrictions. The Public CMA Access Interface will not allow the generation of a skewed configuration, *e.g.*, one that contains both the V1 and V2 implementation instances of the FE shown in Figure 3.

A configuration description can carry a *partition constraint* in terms of names and values of partition attributes. Such a constraint causes the CMA to automatically eliminate all instances as candidates for inclusion in this configuration, whose partition attributes are thus named and have a value incompatible with the value specified in the constraint. In terms of our earlier example, this means that, for a configuration constrained to a "target" partition value "MC68k", an instance with "MC68030" partition value cannot be selected for inclusion.

A configuration is also allowed to have a *rendition constraint* on the instances included in the configuration. It thereby becomes possible to build a configuration containing, for example, "source" renditions only, *e.g.*, for shipping the entire set of sources, but no object code renditions. The specific rule enforced is that instances whose rendition does not specialize one of the renditions specified in the constraint are not eligible for inclusion in the configuration.

Configuration descriptions can be recorded as instances of the logical objects which they configure, and as such they can partake in other configurations. They can also be *expanded* into new configurations to insert their constituents into the latter. Resulting skews are detected and automatically resolved by the CMA through weakening of instance selection constraints and subsequent refinement by the user.

7. *USING THE CMA*

Users can create multiple CMA Data Bases, in which entities and configurations are administered. Each such data base can be subjected to access control restrictions. To resolve naming conflicts upon merger of hitherto unrelated configurations, naming of attributes and objects is done through name spaces, of which there can be an arbitrary number within a CMA data base. These name spaces avoid the need for global name coordination among all users of a CMA data base. Each such name space can be subjected to access control restrictions. Finally, each configuration can be individually access controlled.

The CMA interfaces have a simple transaction mechanism; that is, changes to a CMA data base become visible to other users only after a commitment of the changes. Similarly, changes to a configuration are accumulated in transactions and not made externally visible until an explicit commitment occurs. All transactions can be abandoned to re-establish the most recent committed state.

The main activities for the users and tools interfacing with the CMA are

- first, the conveying of information about logical objects, instances, and their relationships to the CMA;
- second, the gradual and guided build-up of configurations based on entered information; and
- third, retrieving and utilizing of existing configurations.

In building configurations, users have a choice of top-down refinement along component relationships or a bottom-up refinement along logical dependencies, or a mixture of both. At each refinement step, users can impose selection constraints in terms of attribute values without as yet deciding on a particular instance for inclusion in the configuration. For every refinement of a logical object, queries are available that return the set of alternative instances satisfying already existing constraints. These sets can be further narrowed by requesting only those alternatives that lead to configurations of guaranteed consistency. The user can of course back out of previous refinements if necessary. At any time, numerous queries for various aspects of a configuration are possible, such as for the set of unsatisfied dependencies, the set of ambiguous selections, the set of satisfied dependencies of unknown consistency status, the set of causes for the inclusion of a particular instance, and so on.

In practice, the benefits of the CMA become more pronounced with the wealth (and accuracy) of the peripheral tools interfacing with the CMA, and with the re-use of existing sub-systems in a multitude of different configurations.

8. *COMPARISON WITH RELATED EFFORTS*

Configuration management has become an important research topic of software engineering. Many efforts have addressed the problems and produced valuable tools.

Both SCCS [8] and RCS [7] are useful tools for tracking changes of text (source code or documentation) and offer a primitive form of configuration identification. They automate the storage and retrieval of various versions of text files, keep track of when and (optionally) why changes were made and who made them, exercise access control on various versions, and identify configurations in the form of trees of versions. Both delegate the issues of dependency, consistency, compatibility, and generation of new configurations to the user (or other tools).

Make [4] is a tool for the automatic regeneration of a program once some of its modules have been modified. A Makefile can recursively refer to other Makefiles. However, Make has no notion of versions; the consistency, ambiguity, and completeness of a configuration are left to the user to establish or resolve. The development of a new configuration from an old one is a manual operation.

The systems described in [2, 9, 6, 5, 1, 3] are integrated software environments which contain, among others, a configuration management component. The ''configuration threads'' of DSEE [2] are rule-based descriptions of configurations. 'shape' [9] utilizes a much enhanced version of the Makefile [4], and produces a ''configuration identification'' from the selection and variant rules. The Odin system [6] is an extensible object manager. Odin's user defines objects (tools, their input and outputs) using a specification language. Adele's consistency of configurations [1] corresponds to CMA's notion of partitions, and utilizes attributes associated with versions, and constraints on the versions or modules that may be combined together. Gypsy [3] can bind versions to configuration components dynamically, thus allowing configuration families (ambiguous configurations in CMA). Gypsy intentionally allows the construction of skewed configurations, such that skewed versions can be used in building different derived objects. Configuration threads in DSEE, configuration identification in 'shape', derivation graph in Odin, and the target concept of NSE [5] roughly correspond to CMA's notion of a ''complete configuration'', except that none of these systems has the added notion of operationally (in)consistent configurations or of deriving such consistency from other relationships such as compatibility. Odin, Adele, and NSE have a pronounced notion of composite objects similar to CMA. Adele and Odin can dynamically construct new systems, though Odin's form of parameterization is more restricted. NSE, however, has no obvious support for controlling the combination of existing components into new systems.

A common theme to the cited efforts is the automated tracking of configurations as their elements are changed by the user, so that the configured system can be rebuilt automatically. Changes might be minor or whole-sale (via configuration families). An axiomatic assumption in these approaches is that any such change will preserve (or work towards) operational consistency of the configured system. Arguably, this is a good model for the development of a system from scratch, where the configuration is a result of the development, but not an initial input to it. Principles of software reuse partially invalidate this model: systems are no longer built from scratch, but largely configured from pre-existing components. CMA therefore takes the approach of **first** configuring the system by combination of existing (or postulated) components and tools, exploiting known (in)consistencies and (in)compatibilities to narrow the choices, and then permitting other tools to track the development process and report configuration changes and properties back to CMA.

Another difference is the highly generic model offered by CMA. Most of the cited efforts force users into a particular mold of organizing the administrated entities, typically restricted to host files and directories. No such built-in rules exists in CMA; they are left to tools built on top of CMA. Yet, the CMA data model is capable of answering a surprisingly large number of CM-related questions completely independently of such additional rules.

Often, the fundamental capabilities of other CM systems rely on predefined object types that are either non-extensible or difficult to extend by the user, or they require information flow from user-provided tools without offering a clean integration platform. The very nature of CMA is to provide this extensibility and information integration.

Finally, the notion of user-defined attributes with predefined CM-semantics and, in particular, of attribute value specialization and generalization adds considerable expressive power and flexibility to CMA, which we have not found in any of the cited systems.

9. *SUMMARY*

The CMA and its built-in generic concepts are intended to supply a solid foundation for a variety of CM-related tools without a-priori intermingling diverse aspects of application domain, version control policies, configuration management policies, system modelling approaches, change tracking, automated rebuilding, etc. The latter often need to reflect the idiosyncrasies of an organization, programming language, or software engineering environment. In particular, the CMA semantics associated with attributes and their values is quite general. Such semantics, together with CMA's notion of consistency, contributes to the generality and power of CMA's concept of configurations.

By isolating a kernel of policy-independent CM functionality and by resolving the non-trivial implementation choice of a sufficiently general data model, the CMA can serve many tools tailored to these specifics without being merely a general purpose data-base system. Implementation of many such tools or even integration of CMA in a software engineering environment becomes considerably easier, if not trivial. Such implementation or integration allows the user to decide on the desired degree of automation for a given application environment.

REFERENCES

1. J. Estublier. Configuration Management: The Notion and the Tools. Proceedings of the International Workshop on Software Version and Configuration Control, German ACM Chapter, Grassau, FRG, Jan., 1988.

2. D. B. Leblang, R. P. Chase, Jr., and G. D. McLean, Jr. The Domain Software Engineering Environment for Large-Scale-Software-Development Efforts. Proceedings of the 1st International Conference on Computer Workstations, IEEE, Los Alamitos, CA, Nov., 1982.

3. E. S. Cohen et. al. Version Management in Gypsy. Proceedings of the Software Engineering Symposium on Practical Software Development Environments, ACM SIGSOFT/SIGPLAN, Boston, MA, Nov., 1988.

4. S. I. Feldman. "Make--A program for Maintaining Computer Programs". *Software--Practice and Experience 9*, 4 (Apr. 1979).

5. W. Courington. The Network Software Environment. Sun Microsystems Inc., Mountain View, CA, Apr., 1989.

6. G. M. Clemm. The Odin Specification Language. Proceedings of the International Workshop on Software Version and Configuration Control, German ACM Chapter, Grassau, FRG, Jan., 1988.

7. W. F. Tichy. Design, Implementation an Evaluation of a Revision Control System. Proceedings of the 6th International Conference on Software Engineering, IEEE, Tokyo, Japan, Sep., 1982.

8. M. J. Rochkind. "The Source Code Control System". *IEEE Transactions on Software Engineering SE-1*, 4 (Dec. 1975).

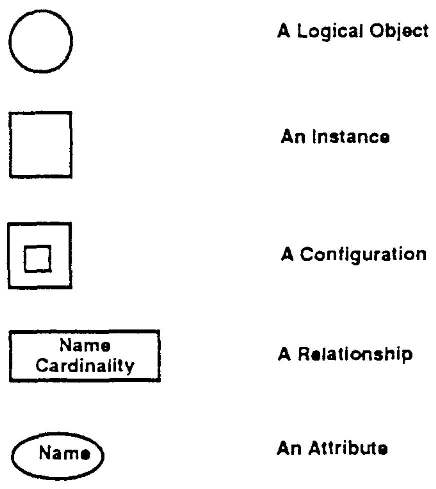
9. A. Mahler and A. Lampen. An Integrated Tool Set for Engineering Software Configurations. Proceedings of the Software Engineering Symposium on Practical Software Development Environments, ACM SIGSOFT/SIGPLAN, Boston, MA, Nov., 1988.

APPENDIX A

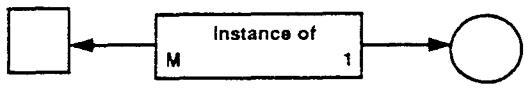
DATA MODEL RELATIONSHIPS

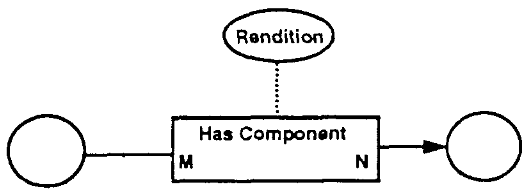
This appendix shows all the relationships of the CMA data model. Each relationship is represented by a rectangle containing the name of the relationship and its cardinality. The arrows emanating from the rectangle of a relationship show the direction of the relationship.

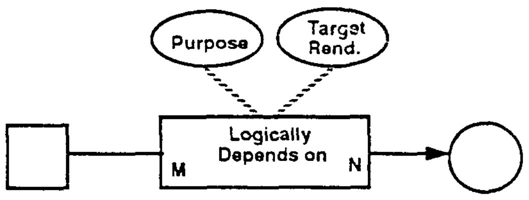
Legend

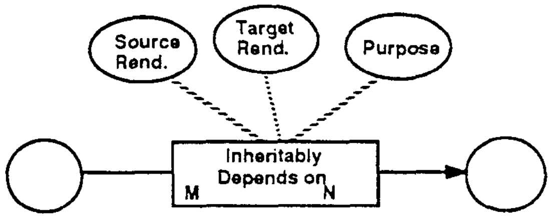


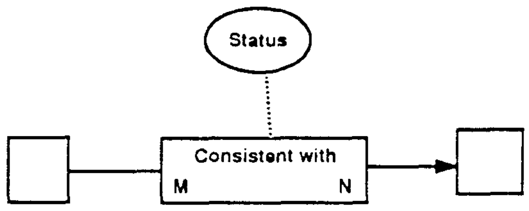
Relationships

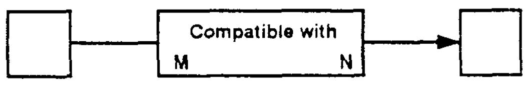

Instance of
M
1

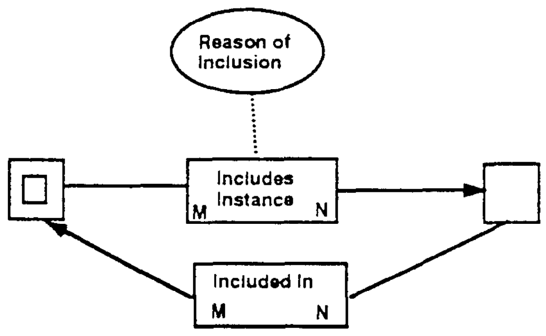

Rendition
Has Component
M
N


Purpose
Target Rend.
Logically Depends on
M
N


Source Rend.
Target Rend.
Purpose
Inheritably Depends on
M
N


Status
Consistent with
M
N


Compatible with
M
N


Reason of Inclusion
Includes Instance
M
N
Included In
M
N