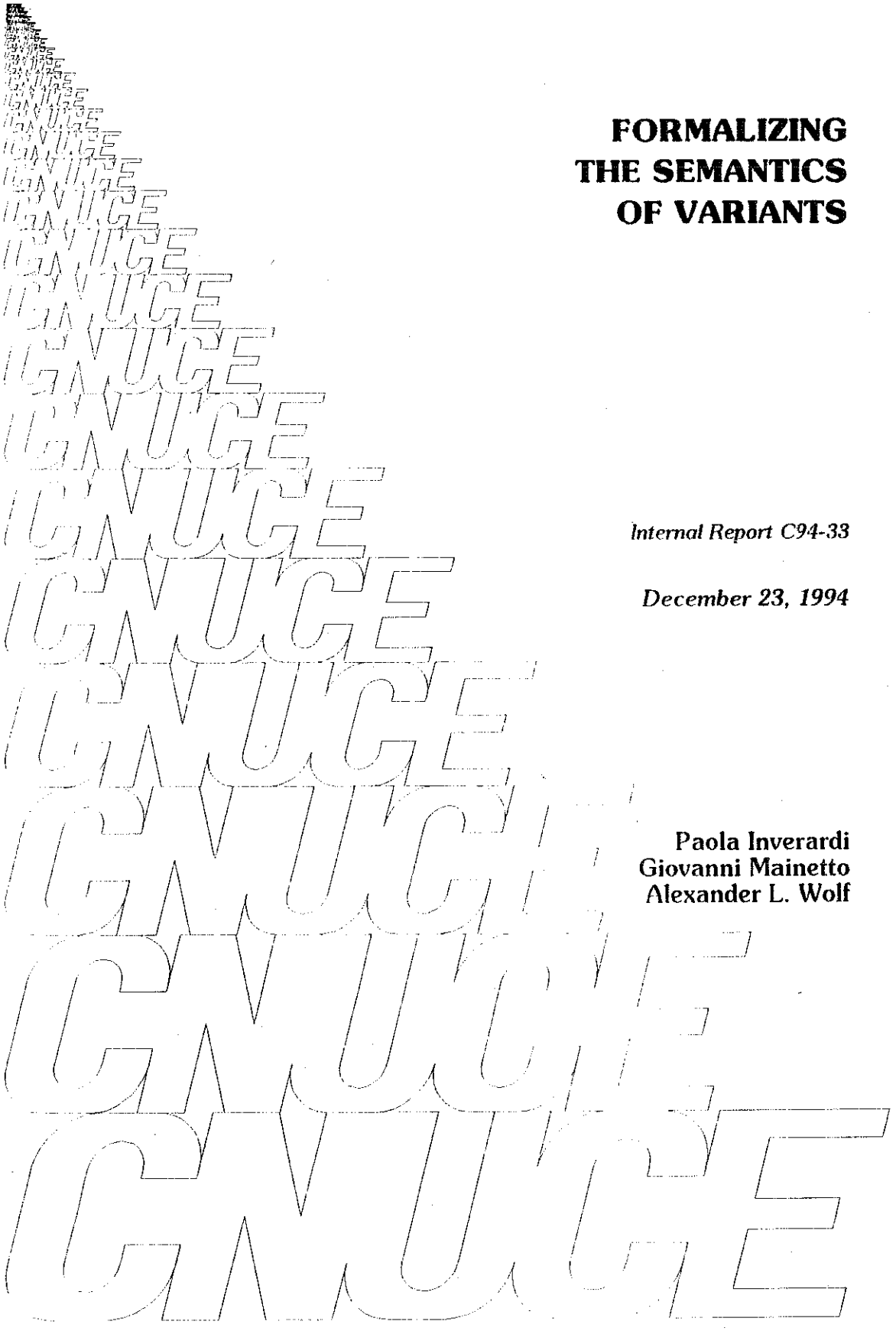

FORMALIZING THE SEMANTICS OF VARIANTS

Internal Report C94-33

December 23, 1994

Paola Inverardi
Giovanni Mainetto
Alexander L. Wolf

Formalizing the Semantics of Variants

Paola Inverardi[♡], Giovanni Mainetto[◇], and Alexander L. Wolf[♠]

[♡]Dip. di Matematica
Universitá di L'Aquila
L'Aquila, Italy

[◇]Istituto CNUCE
Consiglio Nazionale delle Ricerche
I-56126 Pisa, Italy

[♠]Dep. of Computer Science
University of Colorado
Boulder, CO 80309 USA

ABSTRACT

Software Configuration Management (SCM) is the activity of controlling the evolution of a software system. Our goal is to give a precise semantics to SCM concepts in order to form a sound basis for developing improved SCM tools. We are beginning by focusing on the key notion of version. *In particular, we are trying to understand the semantics of* variants*—versions of a configuration item that are equivalent in some context. In this paper we give a formal semantics for variants based on the notion of* preorder relation, *as opposed to the more familiar notion of equivalence. Such an approach allows consistent modifications of interfaces to be taken into account. We also present an initial mapping of those semantics onto an object-oriented persistent programming language that can be used to model an SCM repository.*

This work was supported in part by the CNR, Italy-US bilateral project SIENOSP. A. Wolf was also supported in part the Air Force Material Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the US Government and no official endorsement should be inferred.

1 Introduction

Software Configuration Management (SCM) is the activity of controlling the evolution of a software system. In general, an SCM system should be able to manage a large amount of (heterogeneous) information, to support parallel and distributed software development, and to keep the history of the software product development.

There are two main requirements for an SCM system to be successful.

1. An SCM system should make it easy to perform all the activities it supports. In particular, it should *automate* activities as much as possible.
2. An SCM system should be open and easily integrated into different development environments. That is, it should be made *flexible* by not hard-coding management policies.

These two requirements are clearly contradictory, since automation often implies the need for encoded policies in order to direct application of the basic SCM mechanisms. Nevertheless, it is important to discover ways to reconcile them if we are to make SCM systems more powerful and useful. To do this, we must first identify the fundamental SCM concepts, then define a precise semantics for the concepts, and finally develop support tools that can take advantage of the semantics.

Modern SCM systems typically attempt to reconcile the contradiction between automation and flexibility by making use of the concept of the executable *software process* specification. However, we find this approach to be lacking in the second area, namely a precise semantics for the fundamental SCM concepts. For example, the interpretation of the semantics of relationships among versions is something that now must be left to the user; SCM tools can only record in a syntactic way the existence of some relationship without understanding its meaning. This calls into question the degree to which support tools can now take true advantage of semantics.

In our work, we are attempting to give a more precise semantics to SCM concepts, which should in turn lead to improved SCM tools. We are beginning by focusing on the key notion of *version*. In particular, we are trying to understand the semantics of *variants*—versions of a configuration item that are equivalent in some context. Here, we present a semantics for variants that takes into account consistent modifications to the interface of an item in terms that are expressed by a a *preorder relation* among those items.

To express the semantics of variants in concrete terms usable by SCM tools, we are exploring the mapping of the semantics onto various data models. Here we present a mapping onto one such model, namely that of an object-oriented persistent programming language called OOPPL [7]. In essence, the approach is to make extensive use of type signatures, inheritance, and subtyping. An important aspect of this mapping effort, however, is to make some of the SCM notions *implicit* in the data model by assigning to them a direct correspondence with that model's features. In this respect our approach differs from others described in the literature, notably [5] and [9].

The next section presents our formalization of the semantics for variants. Following that we discuss a mapping of those semantics onto OOPPL. We conclude with a discussion of related work and a summary of the contributions of this work.

2 A Semantics for Variants

To develop a formal semantics for variants, we first need to place the notion of variant within the broader context of other SCM concepts. We then draw the formal semantics of variants in terms of interface congruence.

2.1 SCM Concepts

A software system is a collection of code, test cases, design documents, and the like that constitute the product of a development project. A software system is a member of a system family whose members are strongly related by functionality. A system family is characterized or described by a system model. A system model is used to place a boundary on the system family, giving the parameters that lead to different family members. A family member is called a configuration. The name derives from the act of configuring a system, which is the process of choosing a family member.

A configuration consists of configuration items, which are the artifacts that are under the control (or are visible to) a configuration management system. Configuration items can be hierarchical, which is to say that a configuration item may consist of other configuration items.

A configuration item can be thought of as having an interface that describes its role within a software system as well as its expectations from other items. More generally, a configuration item

has properties that are of relevance to understanding the item and its place within a system. The properties or interface of a configuration item can be described by a type signature. Note that this interface includes the item's functional interface. Moreover, it contains all the information that is relevant for the configuration process as implemented by a configuration program. Thus, it can contain information that is related to the development process, such as machine dependency or asymptotic running time, and not just to the functional behavior implemented in the item.

A configuration item exists in multiple versions. Selecting a family member (i.e., configuring a system) amounts to selecting one version for each item constituting the software system. The notion of version can be thought of as an n-ary relationship among configuration items. In the literature, versions are further distinguished as either revisions or variants. A revision is largely a temporal relationship that records development history.

In contrast to a revision, a variant is an equivalence relationship that records the fact that some set of versions all respect the same (interface) properties. It is, moreover, a congruence relation with respect to configuration contexts—that is, given a configuration program, the same result can be obtained regardless of the particular variant chosen. In other words, all the configuration items that are variants of each other can be considered as equivalent with respect to the configuration that is being built. Note that in this way, configurations assume a deeper semantic meaning. In particular, configurations are now not simply a list of selected configuration items, which is actually a very syntactic view, but instead represent the different observations of a system family that a user can have. Therefore, the list of configuration items that a configuration program gives as result is just one of the possible implementations of the abstract view of the system family as defined by means of that particular configuration program.

2.2 Formal Semantics

We now formalize the concepts introduced above. In this formalization, we try to minimize hypotheses in order to maintain a high degree of generality. In particular, we need only assume that any configuration item exhibits an interface and that interfaces are elements of a set on which two boolean functions are defined: a comparison function and an equality function. This means that given two interfaces it must be possible to determine if they are equal, if one is contained within the other, or if they are non-comparable. If one takes the usual syntactic approach of characterizing

interfaces in terms of the language in which the interfaces are written, then it is enough to choose as the set of interfaces the initial algebra in the set of all algebras with the given signature and trivially define the two boolean functions as syntactic equality and signature containment.

In the following we assume the existence of a set I of interfaces partially ordered by an operator "$<$" that relates two interfaces I_1 and I_2 such that $I_1 < I_2$ if and only if I_1 is contained within I_2. From our point of view it does not matter if this containment relation is syntactically or semantically defined. That is, one could decide to define comparison between interfaces modulo renaming of elements of the signature. We also assume the existence of a set B of bodies that implement some functionality.

We define a configuration item as a pair $\langle I_j, B_j \rangle$, where $I_j \in I$ represents the interface of the configuration item and $B_j \in B$ represents its body. In other words, I_j is the item's type (taking a broad definition of *type* to mean a description of an items properties) and B_j is the item itself. A configuration is then a set of these pairs.

A configuration program is any program that is written in the configuration language and that uses configuration items. By hypothesis we are assuming that configuration programs make use of configurations in order to characterize the (set of) data they operate on, namely the configuration items. We must then assign a semantics to a configuration program and that semantics is simply a function of the set of interfaces used in the program. Therefore, given a configuration program P that uses the set of interfaces $\langle I_1, \ldots, I_n \rangle$, its semantics is a function $f(I_1, \ldots, I_n)$, where f is calculated on the basis of the control structure of P and whose result is a set of configurations. More precisely, let C be the set of configurations (which is obtained by the powerset of configuration items), 2^C be the powerset of C, and 2^I be the powerset of interfaces I. The semantics of a configuration program P is then given by the following function.

$$\mathcal{F}_P : 2^I \rightarrow 2^C$$

In this way we assign to any configuration program a set of possible configurations, thus reflecting the nondeterminism implicit in the notion of variant. Of course, depending on the set of interfaces given to the program, the result can also be just a single configuration.

Although we want to be very general with respect to the definition of configuration program, we must define the following properties in order to precisely characterize the semantics of a config-

uration program with respect to the notion of variant.

Property 1 *Let P be a configuration program and let its semantics be given by $\mathcal{F}_P : \{I_1, \ldots, I_n\} \rightarrow \{C_1, \ldots, C_m\}$. Let c_j be a variant of $c_r \in C_k$, $1 \leq k \leq m$, and let $C'_k = C_k - \{c_r\} \cup \{c_j\}$. Then $C'_k \in \{C_1, \ldots, C_m\}$.*

This property says that the semantics of a configuration program is that of building a set of all the possible configurations, where the number of configurations obtainable are strictly related to the number of variants existing in the environment.

We can now introduce our notion of variant.

Definition 1 *Let $c_1 = \langle I_1, B_1 \rangle$ and $c_2 = \langle I_2, B_2 \rangle$ be two configuration items. Then we say that c_2 is a* variant *of c_1, written $c_1 \preceq_{var} c_2$, if and only if $I_1 = I_2$ or $I_1 < I_2$.*

This definition says that the variants of a given configuration item are all configuration items that either exhibit the same interface or whose interface contains the original one. Note that in this way we are able to consider variants also as configuration items that have a different—but still consistent—interface. Therefore, we enlarge the informal semantics of variant presented in the previous section, and as a consequence we are forced to rely on formally a weaker (but more appropriate) relation than an equivalence, namely a preorder relation.

It is easy to verify that the variant relation defines a preorder on the set of configuration items.

Fact 1 *$\preceq_{var}$ is a preorder on the set of configuration items.*

Recall that a preorder relation is a relation that is reflexive and transitive, but not in general symmetric. Given our definition of variant, this is rather sensible since we can consider a configuration item equivalent to another if their interfaces are equal or if one is a subinterface of the other; in the latter case, the symmetric property obviously does not hold.

The next step is to show that if we take configurations as the contexts of interest, then this notion of variant actually allows the definition of a *context preserving* relation on the set C of configurations. This means that our notion of variant permits a configuration program to preserve its behavior regardless of the specific configuration item variant used. Of course, this is exactly the meaning of variants that we wish to capture.

Definition 2 *A preorder relation $\preceq$ is* context preserving *if for any context $C[\bullet]$, $t_1 \preceq t_2$ implies that $C[t_1] \preceq C[t_2]$.*

In the case in which the preorder relation is actually an equivalence relation, the above definition corresponds to the definition of *congruence*.

We are interested in considering configurations as contexts and therefore give a definition that extends a preorder relation defined on a set S to its multiset 2^S.

Definition 3 *Let $\preceq$ be a preorder relation on a set S. Then $\prec\preceq$ is the multiset preorder defined on S in the following way. $\{s_1, \ldots, s_n\} \prec\preceq \{r_1, \ldots, r_n\}$ if and only if $s_1 \preceq r_i$ for some i, $1 \leq i \leq n$, and $\{s_2, \ldots, s_n\} \prec\preceq \{r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n\}$.*

Let us now consider the multiset extension of the variant preorder $\prec\preceq_{var}$. It defines a preorder relation on configurations relating them in the following way.

Proposition 1 *If $C_1 \prec\preceq_{var} C_2$, then for any configuration program P such that $\mathcal{F}_P = R$, $C_1 \in R \Rightarrow C_2 \in R$.*

In other words, the relation holds between pairs of configurations that are in the same image of a configuration program.

Proof 1 *If $C_1 \prec\preceq_{var} C_2$, then by definition of $\prec\preceq_{var}$ it holds that the two configurations differ because of some configuration item that is in a variant relation with a corresponding one in the other configuration. By the property of configuration programs, we get the thesis.*

Note that the reverse does not in general hold. If two configurations are in the same image of a configuration program, it is not in general true that they are in the $\prec\preceq_{var}$ relation. The two can in fact differ because of incompatible configuration items that are respectively variants of other configuration items.

We have now defined over the set of configurations a preorder relation that allows us to consistently compare configurations. Our ultimate goal is to show that the notion of variant introduced above preserves configuration contexts. In the following, we use the notation $C[\langle I, B\rangle]$ in order to identify a configuration that contains the configuration item $\langle I, B\rangle$, where $C[\bullet]$ represents the context—i.e., the balance of the configuration items constituting the configuration excluding $\langle I, B\rangle$.

Proposition 2 *Let $\langle I_1, B_1\rangle \preceq_{var} \langle I_2, B_2\rangle$ be two variant configuration items. Then for any configuration $C_0[\langle I_1, B_1\rangle]$ it holds that $C_0[\langle I_1, B_1\rangle] \prec\preceq_{var} C_0[\langle I_2, B_2\rangle]$.*

Proof 2 *By definition of $\prec\preceq_{var}$.*

This proposition shows that $\prec\preceq_{var}$ is a context preserving relation.

Corollary 1 *Let* $\langle I_1, B_1\rangle \preceq_{var} \langle I_2, B_2\rangle$ *be two variant configuration items. Then for any configuration* $C_0[\langle I_1, B_1\rangle]$ *that contains* $\langle I_1, B_1\rangle$*, it holds that for any configuration program* P *such that* $\mathcal{F}_P = R$*,* $C_0[\langle I_1, B_1\rangle] \in R \Rightarrow C_0[\langle I_2, B_2\rangle] \in R$*.*

3 A Mapping to an Object-oriented Data Model

As discussed above, we can think of expressing the notion of variant as a type that defines the interface of the configuration item to be versioned. Through the type, it defines its relationship to the external world. The idea is then to use the interface as the type of all equivalent configuration items. Very roughly this means that for any configuration item for which there are variants, a new object class is created that defines the interface of the configuration items and we start populating that class with the bodies of the given item. When one wants to create two development lines of variants—for example, two variants of the same system for two different machines—we can perform the same process by creating two subclasses that enlarge their interfaces with the additional piece of information related to the machine. We show an example of this below using a particular object-oriented data model.

3.1 An Object-oriented Data Model

The Object Oriented Persistent Programming Language (OOPPL) [7] is a (non-pure) functional persistent programming language having object-oriented features [3, 4]. OOPPL is an evolution of the Galileo language [1] that preserves the most relevant database constructs provided by the original language. It adds classes with multiple inheritance (*is_a* relation) and objects.

An OOPPL class is a bulk type constructor, like *set_of* or *list*, with both an intentional and an extensional meaning. The intentional meaning is that all objects belonging to a class have a minimal type that is a subtype of that specified when a class is defined. The extensional meaning is characterized by two facts: an *instance_of* relation exists between a class and the set of actual objects belonging to it (this set of objects is called the extension of the class) and a subset relation exists between the extensions of two classes that are in an *is_a* relation. Thus, there exists simultaneously a subset relationship between the subclass extension and superclass extension, and

a subtype relationship between the type of the subclass objects and the type of the superclass objects. The class mechanism allows the programmer to work on the extension of a superclass by inspecting all its objects, including those that are members of its subclasses, but only using the properties specified in the superclass.

OOPPL supports overriding, late binding, and migration of the same object between classes if they are in *is_a* relation. Objects are extensible records and every object has a unique object identifier. The same object can simultaneously belong to several classes, but only those classes that are in the *is_a* relation. Equality on objects means sameness: two objects are equal if and only if they have the same object identifier. The identity of an object is used for modeling its association with other objects (e.g., aggregation). Objects can have values of any type as components, such as, for example, functions used to model methods of classical object-oriented languages.

OOPPL is a useful programming language for the definition of a database schema and the retrieval and manipulation of persistent values. For our purposes, this means a database of configuration items that exist in some version relationship. The definition of the database schema is performed through a sequence of top-level declarations.

OOPPL is a strongly-typed language. Moreover, OOPPL type checking is almost exclusively static (i.e., performed at compile time) although there is some allowance for dynamic (i.e., run time) type checking. OOPPL is also a polymorphic language that supports, in particular, inclusion polymorphism. Support for inclusion polymorphism has the following meaning. If S is a subtype of T, then a value of type S can be used in every context (i.e., in every sentence of the language) where it is possible to use a value of type T. On the contrary, it is forbidden to use a value of type T in every context where it could be used as a value of type S. The support of inclusion polymorphism is a fundamental property of a language that wants to provide a class data type and inheritance. We notice that inclusion polymorphism is a well-defined notion of congruence, and thus OOPPL has a "built-in" notion of congruence that is very useful for SCM purposes.

3.2 An Example

In order to illustrate our approach, we present an extremely simple example of how we intend to express the notion of variant in terms of the object-oriented features of the data model. We define a set of configuration items that specify input-output devices and the (corresponding) access methods.

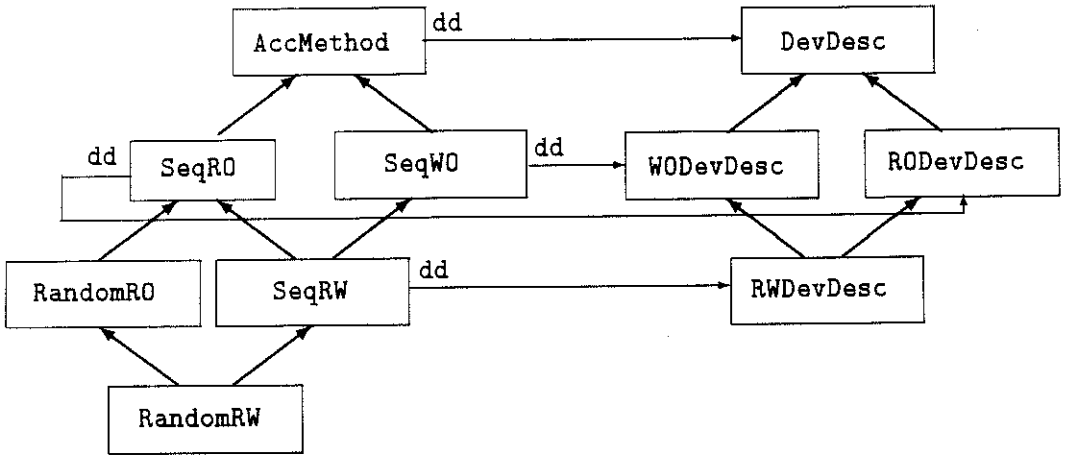

Figure 1: Example Type Hierarchy of Variants.

Depending on the access modality, we distinguish among various kind of access methods, such as sequential read-only (`SeqRO`), sequential write-only (`SeqWO`), sequential read and write (`SeqRW`), random read-only (`RandomRO`), and so on. Analogously, we differentiate devices depending on their characteristics, such as, for example, write-only device descriptors (`WODevDesc`) or read-only device descriptors (`RODevDesc`).

We introduce a class definition for any set of configuration items that are variant. The configuration items are then modeled as instances of the given class. Figure 1 shows a graphical representation of these relationships, while Figure 2 shows their translation into OOPPL. For example, the class `WODevDesc` is populated with all the configuration items that define sequential write-only devices as printers. Using the subclass mechanism, all other class definitions can be consistently related to each other whenever sensible. The subclass relationship is indicated by the heavy arrows in Figure 1.

The OOPPL data model also allows functional dependency to be taken into account by means of the type mechanism. In particular, if a class definition uses the name of another class, then this implicitly determines a (functional) dependency between the two classes. The dependency relationship is shown as the light arrows in Figure 1.

Note that in the OOPPL declarations of Figure 2, we have been able to refine the dependency

```
let DevDesc: class [
        status: integer,
        .....];
     WODevDesc: subclass DevDesc with [...];
     RODevDesc: subclass DevDesc with [...];
 RWDevDesc: subclass WODevDesc,RODevDesc with [...];

    AccMethod: class [
        dd: DevDesc,
          next: integer,
        open: ()->void,
        close: ()->void];
     SeqRO: subclass AccMethod with [
dd: RODevDesc,
read: ()->char];
     SeqWO: subclass AccMethod with [
dd: WODevDesc,
write: (char)->void];
     RandomRO: subclass SeqRO with [
seek: (integer)->void];
     SeqRW: subclass SeqRO,SeqWO with [
dd: RWDesc];
     RandomRW: subclass RandomRW,SeqRW with [];
```

Figure 2: OOPPL Declaration of Variants.

relation by using the definitional mechanism of "overridding" on the imported device. This allows for specializing the dependency consistently with the superclass definition. For example, in the subclass definition `SeqWO`, the definition `dd: WODevDesc` is correct only because `WODevDesc` is a subclass of `DevDesc`.

A configuration program can make use of the declarations to reason about acceptable configurations. For example, suppose that at some point there is a configuration item of (immediate) type `SeqRW`, but no configuration item of (immediate) type `SeqRO`. Suppose further that the user requests, through the configuration program, a system that requires an item of type `SeqRO`. Notice that this request implies an additional request for an item of type `RODevDesc`. The configuration program can satisfy the request—despite the fact that there is no instance of type `SeqRO`—by selecting the instance of type `SeqRW`, since it is a subclass of `SeqRO`. Moreover, since it has selected the instance of `SeqRW`, it automatically can select a correct device descriptor, namely an instance of `RWDevDesc`.

4 Conclusion

Numerous SCM systems make use of databases, including object-oriented databases, as repositories for configuration items (e.g., [5, 6]). Ambriola and Bendix [2] describe an object-oriented approach to configuration control. They use the object-oriented modeling concepts to maintain configuration information about modules, but do not try to capture the semantics of version relationships.

Perry [8] describes an approach to formalizing the notion of version using the Inscape environment. His approach differs from ours in that he develops an axiomatic semantics that requires special reasoning facilities going far beyond object-oriented type inference.

Our approach is to give a precise semantics to versions, based on the use of a preorder relation among configuration items, and then to map those semantics onto an object-oriented database. The approach needs extensive experimentation, but the small example of the previous section demonstrates its promise. We can already see two immediate effects. One is that a certain amount of the complexity of SCM is reduced, since no explicit notion of variant and of dependency has to be taken into account, but instead can be made implicit in the support structure. The other is that

a certain degree of automation is reached, thus helping to promote consistency among the variants.

Both effects result directly from the formalization of the semantics and the mapping of onto an object-oriented data model. The formalization, of course, restricts the syntactic notion in some way—that is the point. In particular, we use the idea that variants are implementations of the same interface, within the framework of subtyping. Subtyping, in fact, allows us to describe all those modifications to the interface that still maintain some degree of semantic consistency with the original. Moreover, dependency relations are automatically taken into account and consistent refinement of these relations are supported.

The class approach also suggests the idea that a system family can actually offer different observations of its member systems at various level of abstraction. For instance, referring back to the example above, at a certain point of the development process, the user might not be interested in considering specific devices and/or specific access methods, but only want a consistent configuration in order to test a part of the system that does not depend on the differences among I/O functionalities. This can be straightforwardly expressed by simply asking for an instance of the top class `AccMethod`. Another argument in favor of this modeling approach is that it is easier to keep the *functional* dependencies separate from the *development* dependencies—that is, all those dependencies among configuration items that are due to the software production process. In fact, the functional dependencies are automatically supported and do not need to be explicitly recorded in the database.

REFERENCES

[1] A. Albano, L. Cardelli, and R. Orsini. Galileo: A Strongly-Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[2] V. Ambriola and L. Bendix. Object-oriented Configuration Control. In *Proceedings of the Second International Workshop on Software Configuration Management*, pages 133–136. ACM SIGSOFT, 1989.

[3] M.P. Atkinson, F. Bancilhon, D. Dewitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-oriented Database System Manifesto. In *Proceedings of the International Conference on Deductive and Object-oriented Databases*, pages 40–57, 1989.

[4] M.P. Atkinson and O.P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[5] J. Estublier and R. Casallas. The Adele Configuration Manager. *Trends in Software*, 1(2):99–134, 1994.

[6] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented Version Description in EPOS. *Software Engineering Journal*, 6:378–386, 1991.

[7] G. Mainetto. Static and Dynamic Semantics of OOPPL. CNUCE-CNR, December 1994.

[8] D.E. Perry. Version Control in the Inscape Environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149. IEEE Computer Society, March 1987.

[9] A. Shore. Versioning and Configuration Management in an Object-Oriented Data Model. *Very Large Database Journal*, 3(1):76–106, January 1994.